RICE UNIVERSITY

Cache Coherence Using Local Knowledge

by

Ervan Darnell

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Ken Kennedy,
Noah Harding Professor of Computer Science

Mark Krentel, Lecturer of Computer Science

John Bennett,
Associate Professor of Electrical Engineering

John Mellor-Crummey,
Faculty Fellow of Computer Science

Elana Granston,
Research Scientist of Computer Science

Houston, Texas

April 1995

ABSTRACT

Cache Coherence Using Local Knowledge

by

Ervan Darnell

Hiding memory latency is critical in modern machines. Typically, machines have used cache and addressed the ensuing cache coherence problem with hardware or VM-based strategies that rely on global inter-cache communication. However, global communication limits scalability. "Local knowledge" coherence strategies, which use compile-time information to avoid run-time global communication, offer better scalability, but suffer additional cache misses. We develop a framework for understanding the relation of coherence strategies, previous and newly proposed. Within this framework, it is possible to define, independent of implementation considerations, an "ideal" local strategy with respect to cache hit rate. No local strategy could ever do better. For Fortran programs with readily analyzable subscripts, ideal local strategies achieve the same hit rates as global strategies.

We develop three new local coherence strategies, CTV, TS1, and TS', designed to exploit minimal, aggressive, and reasonable hardware support, respectively. CTV is suitable for machines with no hardware assistance for cache coherence except the bare minimum of an exposed invalidate instruction. TS1 implements the abstract theorems of ideal local coherence as a concrete algorithm. Though the implementation is probably too expensive for a real implementation, TS1 is a vehicle for studying the limits of local coherence. TS' treats coherence over array sections as a graph coloring problem. So long as there are sufficient colors (realized as bits per cache line), TS' is an ideal local strategy. We found that four colors are adequate for many programs. When more colors are needed, TS' degrades gracefully. Its execution overheads are negligible and its hardware implementation costs moderate.

Our data shows that TS' has better hit rates than the best previous local strategy, time-stamping, for nearly all programs, and thus better expected performance. Our data also shows that TS’ achieves hit rates equal to global strategies for analyzable programs, and nearly so for partially analyzable programs. We indirectly compared the performance of TS’ and a particular VM-style global strategy. TS’ has better expected performance on our test suite. For machines without global coherence hardware, local strategies are an effective approach for an important class of programs.

Acknowledgments

Foremost, I would like to thank my thesis advisor Dr. Ken Kennedy for his guidance and his steadfast belief in my ability, even when I doubted myself. He provided the time to think when I needed it and the pressure to perform when my motivation ebbed. Despite our disagreements, I would like to thank Dr. John Mellor-Crummey for his willingness to become involved in the details of my research and for the many constructive criticisms he has given me, both with respect to the technical issues and with respect to getting published. In this final year, I have much appreciated the time Dr. Lani Granston has given me, suggesting which details were the most important to resolve, puncturing holes in flawed new ideas, and helping me to collect the pieces into a presentable whole.

Next, I credit my parents for having quietly instilled, but fervently believed, the idea that academic success is important. It matters primarily for gaining the self-respect that comes from a achieving a difficult goal by your own effort. But it is also significant in my case for showing that someone can move beyond their humble origins regardless of expectations.

I have greatly enjoyed the friends I made in Houston and the fun times we have spent together. I want to thank them for helping me keep my spirits high during those times when the research did not compel my attention. I especially want to recognize my fellow bike team members, Jerry D'Ambrosio, Karl Wagner, & Johnny Roberts, for helping me realize that I could be athletically as well as mentally competitive. There are many more people, too many to list, who deserve credit for having provided quality intellectual conversation outside of the field of computer science, especially in philosophy and politics, and who shared the best of occasions, sailing, camping, and 'Road Tripping'.

Table of Contents

Tables

Figures

Theorems and Definitions

Chapter 1
Introduction

As a matter of economics, if not physics, fast storage costs more than slow storage. Application size drives memory size, and indirectly memory speed. In contrast, processor speed does not need to be compromised because of the application size. Thus, on real machines, memory will tend to be slower than the processor.

Shared-memory multiprocessors exacerbate this problem in two ways. First, the path from memory to processor grows longer with increasing numbers of processors, even as the memory per processor stays the same. Second, simple networks suffer from contention, one processor waiting on another to release the path to memory. Contention can be partially relieved with multiple access paths, but only at the cost of adding more switching delay.

The result of these effects is that processors can consume data faster than memories can produce it. In modern machines, accessing memory can often require 50 processor cycles [17,19]. The current trend is that this gap is increasing. Such large memory latencies represent a serious performance problem that must be addressed.

We introduce the major approaches to hiding memory latency here and cover them in more detail later. The three major ways of hiding latency are:

- Overlapping fetches and computation
- Static data layout
- Caching

Overlapping hides memory latency by moving fetches so that they have more time to complete before the data is needed. It has two problems. First, the total bandwidth requirements are *at least* as large as when doing nothing to address latency. Bandwidth may already be the limiting factor on performance in many cases. Second, the processor and memory system may not support a sufficient number of simultaneously outstanding prefetch requests.

Static data layout techniques (e.g. SPMD) attempt to lay out data for non-uniform memory access (NUMA) multiprocessors so that it is near the processor that uses it. The difficulty is that it may not be possible to tell which processor that is, either because the dependence pattern is not analyzable or because dynamic load balancing is being used.

Caching dynamically replicates values and moves them closer to processors where the access latency is much less. Unlike prefetching, caching generally *reduces* bandwidth requirements. Caching primarily uses special hardware to make run-time decisions about where to place data based on recent behavior. Unlike static layout techniques, caching can be effective even when the dependence pattern is unanalyzable. For these reasons, caching is certainly the most widely used and the most generally applicable.

Unfortunately, caching has a serious drawback in multiprocessor environments: *coherence*. Multiple copies of a logically single data item must be *consistent*, i.e. appear to have the same value on different processors at the "same" time, at least from the application's perspective. Most approaches to coherence are not completely *scalable*. As the size of the problem and the number of processors grows, the coherence cost increases faster still. Many current machines have hardware coherence protocols that are efficient for some fixed number of processors, but they cannot be readily expanded. Protocols suitable for an arbitrary number of processors are also available. For them, the cost of coherence is a significant overhead.

Scalability is a problem when all of the coherence decisions are made at run time because extra information must be communicated globally between processors. Compilers can address this problem by making coherence decisions at compile time. In effect, the compiler tries to predict what coherence operations a purely run-time strategy would perform and adds those to the code, thus substituting compile-time for run-time analysis. The remaining run-time coherence work can be *scalable* because all of the directives (and their supporting hardware) have purely local effects. All of the non-local information has been estimated at compile time.

However, since compile-time analysis must be conservative, there are programs for which some valid values will be unnecessarily removed from the cache, thus reducing hit rates. For *analyzable* programs (those for which the compiler can accurately predict write accesses) this is not the case. Compiler-assisted strategies can preserve all of the reuse that run-time strategies do, for all schedules.

This thesis is concerned with the usefulness and limits of compiler-assisted strategies. It presents a theoretical framework for evaluating and classifying coherence strategies. It explores three new compiler-assisted approaches, each suitable for a different degree of hardware support, which are compared to existing approaches using the framework developed here. Experimental results are presented that demonstrate the impact of the distinctions made in the framework. In particular, analyzable programs, which include many important computational kernels, can be executed in a scalable fashion with no cache misses beyond those logically necessary.

The remainder of this thesis is organized as follows. In chapter 2, we discuss previous approaches to hiding memory latency. In chapter 3, we present a machine model (§3.1) and background definitions that are necessary for understanding cache coherence. With these, we are able to more formally define *coherence* (def. 3.4) and develop our algorithmic framework (§3.2) for analyzing coherence strategies. In chapter 4, we survey previous compiler-assisted methods in more detail. These methods are compared and put in the framework.

In chapters 5 and 6, we discuss our strategy Coherence Through Vectorization, CTV and CTV+, methods suitable for machines with a minimum of hardware support. In chapter 7, we discuss TS1, an idealization of how compiler-assisted strategies would perform without hardware cost constraints. In chapter 8, we discuss TS’, an aggressive, but plausible, compiler-assisted strategy that improves on previous compiler-assisted strategies. In chapter 9, we discuss our experimental results. In chapter 10, we present our conclusions.

Chapter 2

Approaches to Hiding Memory Latency

This chapter describes in more detail the previously introduced approaches to tolerating memory latency . We consider the non-cache approaches, then the cache approaches. The latter are shown to address the problems of the former but introduce their own new one: scalability.

2.1 Non-Cache Approaches

2.1.1 Overlapping Fetches and Computation

One class of approaches to hiding latency is to issue fetches in such a way that they can be performed simultaneously with computation on previously fetched data. This requires predicting which values will be needed later in the computation in order to keep the processor busy.

One approach is to switch between different operations that can be done simultaneously. In this approach, one can view the address decode cycle for a given instruction as the prediction and the execute cycle of the same instruction as the computation. The execution of the complete instruction then takes at least as long as fetching a memory word and several instructions are being executed in parallel to keep the actual processor busy. The parallelism can come at all of these levels. It can be instruction-level parallelism (i.e. superscalar processing). It can be inter-process or inter-task parallelism. The Tera [5] is such a machine, using all of these levels as well as some ones intermediate between instruction level and inter-task level ('futures'). We refer to all levels above instruction-level parallelism as *threads*.

For the processor to be doing useful work between the decode and execute, there needs to be sufficient parallelism to keep the processor busy. For example, a machine with memory 50 cycles away and an average of 5 cycles of computation per memory reference would require 10-fold parallelism to cover all of the latency. Typical programs have 2- to 4-fold instruction level parallelism [37]. Thus there is not sufficient instruction level parallelism in most programs. Therefore, this approach can only work by having substantially more thread-based parallelism available than there are processors. That further requires the processor to be able to switch tasks in only a few cycles if it is to take advantage of fetches in other tasks.

The Tera fulfills these conditions by having separate register sets and memory buffers for each task running on a given processor. Whenever one thread stalls because of a memory access, the processor quickly switches to another thread (or task) for which the data is now available.

However, there are several difficulties with this scheme:

- There must be sufficient available parallelism.
- The total memory bandwidth must be as great as the total processor bandwidth.
- The bandwidth problem becomes worse as machines grow.
- Commodity CPUs cannot be used.

The first problem is that if there is not sufficient available work, the processor will be left waiting on memory. There needs to be sufficient work at every moment, not just on average. For instance, if there is sufficient work available on average only because of multiple tasks within a single process, *p*, there might be insufficient work available in other processes during serial sections of *p* to keep the processor busy.

The second problem is that hiding latency by thread switching does nothing to lower total bandwidth requirements. Every apparent global memory access in the program requires a word to traverse the memory network (instead of being found in a local memory or cache). To preserve the effective low latency, it is necessary that the memory network have sufficient bandwidth to feed all of the processors at their expected maximum data rate. This problem exists at each memory bank as well as for the memory network as a whole. One memory bank could cause delays because of contention for it, even though there is sufficient bandwidth overall.

The third point is that this situation gets worse as the machine gets bigger. More memory and more processors will tend to make memory further away and require more parallelism (per processor) to continue to hide the latency.

The other approach to overlapping fetches and computation is to predict what data will be needed within the same thread and prefetch it [54]. Unlike the thread switching approach, this requires some assistance from either the programmer or the compiler to explicitly compute the address of a word before it is needed. This is more useful (and easier to implement) for loops than straight-line code. For instance, consider the following fragment of Fortran code

```
DO I
   A(I)=B(I)+1
   ...
ENDDO
```

This loop could be transformed to:

```
DO I
   Prefetch (B(I+1))
   A(I)=B(I)+1
   ...
ENDDO
```

`B(2)` is requested on iteration `I=1` and hopefully available before it is needed in iteration `I=2`. It may be made available by reserving a register at the time of the prefetch and loading it there. Typically it is simply loaded into cache and acts only as an advisory instruction. The actual load hopes to find the prefetched value in cache but does not require it to be there. If the caching approach is used, it is still necessary to address the issue of cache coherence. However, if prefetching is effective in hiding most of the latency, cheap and simple coherence protocols may be appropriate. The problems with prefetching are similar to those of using parallelism to cover latency:

- Accesses must be sufficiently predictable.
- Without caching, total memory bandwidth must be as great as the total processor bandwidth.
- Covering all of the latency becomes more difficult as machines grow.
- Prefetching requires special caches or buffers to avoid lockups.

The second and third points are true for reasons identical to the ones already listed for the thread-switching approach. The first point is a similar concern to needing enough parallelism in the thread switching approach. For prefetching, prediction needs to be able to look far enough ahead that it can prefetch early enough for the data to arrive before it is needed. For loops, this is usually easy because it is a simple matter to look back as many iterations as necessary to cover all of the latency.

The more interesting question is about where to store these values. If the degree of necessary prefetching is fixed at some small constant, it is reasonable to allocate registers to hold the values when they arrive. But, when longer look-ahead distances are needed, it becomes necessary to use a cache. The first reason is simply that there are too many values being simultaneously prefetched to allocate registers. The second, more important reason is that much prefetching will be speculative. There needs to be some way to release a prefetched value if it is never used. Handling this explicitly can be difficult. Cache solves the problem simply by evicting the values when the space is otherwise needed. With caches again comes the need for coherence. However, if prefetching is effective at hiding latency, the coherence strategy can be simple.

In summary, overlapping fetches and computation either by prefetching or parallelism can hide latency but it does nothing by itself to lower total bandwidth requirements. Larger machines will require increasingly faster memory networks for overlapping to remain effective. Caching addresses this problem by keeping data close to the CPU. Furthermore, as the trend of memory getting further away from CPUs continues, the degree of overlapping has to continue to grow to remain effective. For the prefetching approach, this indicates that a cache is needed regardless to store the prefetched values. The parallelism approach can avoid using cache only so long as there is sufficient parallelism available and the cost of duplicating register sets is reasonable.

2.1.2 Static Data Layout

Another approach to hiding latency is to specify a static layout by carefully controlling the initial placement of data, the scheduling, and the communication, as in High Performance Fortran (HPF) [41]. The programmer, or possibly the compiler, specifies the distribution of array data with respect to processors by using special program directives. The compiler then optimizes the data communication for an SPMD model (appendix D). The assumed model is that all memory is distributed in some fashion so that each processor has some local memory and the rest is accessible with greater latency. There is no automatic address management, inclusion property, or automatic eviction as there is with a hardware cache. All addresses (including those of apparently remote references) must be resolved as local addresses at compile time. Static layout techniques address the problem of coherence by putting every data word in a single known location, or having the compiler explicitly manage the multiple copies that it creates as separate items in a different address space. There is nothing to be kept coherent in the sense of having multiple copies representing a word at the same address.

The purpose of static data layout is to statically predict which processor will use which data and then place the data close to that processor before the program begins. This is in contrast to the prefetching approach, which tries to dynamically predict where data will be needed and to move it close to the processor just before it is actually referenced.

The obvious advantage of the static approach is that expensive communication is often avoided. What would be main memory references in a shared-memory system can be proven local at compile time. Another advantage is that when communication is necessary, it can often be predicted ahead of time and the benefits of prefetching can still be realized by having the owner send the data directly to the referencing processor. Static data layout looks abstractly like prefetching at compile time. It works at a high level and can move logical units of data (e.g. a whole row of an array) at once instead of moving individual cache lines as shared memory systems usually do. For machines with high message start-up costs this can be an important advantage. The disadvantage is that if data reference patterns cannot be accurately predicted at compile time, the run-time overhead to resolve the memory references will be expensive.

The problem for static data layout comes in handling those accesses which are known to be remote or simply not known to be local. The first problem is that the original static layout may be poorly chosen for actual run-time conditions. This can cause unnecessary communication. This is especially true for irregular problems, where the data distribution changes as the algorithm progresses. Static data layout also effectively prevents dynamic load balancing where the run-time location of data is unknowable at compile time. For static layout techniques to be effective, the compiler needs to be able to accurately predict the amount of computation as well as the data dependence.

Even without load balancing, the address space can become fragmented in which case resolving addresses becomes a significant problem. For a truly distributed memory machine (DM) , local addresses appear as normal addresses, but remote addresses need to be handled with send and receive calls. The sender needs to know what to send, the receiver needs to know what to receive, and both need to agree on the message sequence; there is no request and reply. For analyzable computations, it can be known at compile time which references will be remote, which will be local, and what their addresses will be. For those references which are not analyzable this obligates both the sender and receiver to loop over the whole index space of the computation, not just the part of the index space that a given processor does its computation on. This happens before the actual computation, to decide which references require communication and to which processor they must be sent (or received from) [52]. This requires extra overhead just for the pre-loop that handles the communication, but the advantages of prefetching can be preserved. Many optimizations are possible on the communication loops, but this resolution process will still be expensive in many cases. In the worst case, the value of the parallelism will be capped at some fixed amount because each processor must perform part of the computation for all of the index space.

In summary, static data layout techniques are designed for distributed memory systems where some memory is local to each processor, the rest is further away, and hardware does nothing to automatically move data. An attempt is made to initially allocate data near the processor that will use it. This can lower communication costs when:

- The dependence pattern is analyzable at compile time.
- A layout can be determined which puts data near the processor which needs it.
- The data does not need to change its location an indefinite number of times.
- There is no need for load balancing.

Static data layout techniques become more expensive as these assumptions break down because extra run-time code must be used to determine the current location of data. Caching addresses all of these concerns by deferring as many decisions as possible until run time. The problem with caching is that it must pay the price of maintaining coherence. The second problem is that handling another layer of indirection for ultimately local references is more expensive (in hardware cost if not time) than knowing *a priori* they will be local. This thesis addresses the first issue by borrowing some methods from static layout techniques and developing some new methods that move the coherence cost from run time to compile time. The second problem requires machines that handle both kinds of access in hardware. This thesis closes (chapter 10) with some speculation on integrate the best of both approaches within a single program.

2.1.3 Distributed Shared Memory (DSM)

Distributed shared memory machines (DSMs, such as. the Cray T3D [19]) logically extend the address space across all processors. Instead of needing to use sends and receives as in a DM approach, the memory hardware routes access to the proper processor's node. Data on a different processor's node is still further away and not in a contiguous address space. Using the running example of block distributing `A(1000)` between 10 processors, it will not be the case that &`A(100)`=&`A(101)`-1. Like the DM case, these addressing considerations need to be explicitly dealt with and the run-time mechanisms are similar. Unlike the DM case, a DSM can perform the communication just by calculating &`A(101)` and then requesting that address.

On a DM system, the sender needs to examine every address that possibly needs to be sent. DSMs address this problem in part by letting the receiver request the data it needs simply by issuing the address and letting the memory hardware resolve it and fetch the needed value. For uncertain dependence patterns, this eliminates the cost of the sender checking what it might need to send. In the DM model, the sender might examine more addresses for a possible send than it ultimately communicates. Using the DSM approach of receiver (referencing processor) requests solves this. These requests can also be issued as prefetches to regain the value of prefetching. However, doing so relinquishes two benefits of the pure DM approach. First, it requires that a each datum fetched use a request and a reply. In a pure DM approach, a single message moves the datum. Second, DM techniques, by explicitly recognizing the data to be put in a message, can aggregate that data and save communication overheads. DSM techniques tend to work a word at a time thus paying the startup and transport costs for every word, instead of amortizing the message startup costs.

However, DSMs are similar to DMs in that they both suffer from the costs of needing to run-time resolve addresses, and we consider both to be static data layout techniques.

2.2 Cache Approaches

Caching addresses latency by using temporal and spatial run-time locality to predict access patterns. When a word is accessed on a processor, it is assumed that that word, or another one near it, will be accessed on the same processor again in the near future. Caches are implemented so as to leave a cache-line-sized set of words 'near' the referencing processor, in its local cache.

From this perspective, main memory can be seen as a cache for virtual memory (VM) backing store and cache memory can be seen as a cache for main memory. Managing cache (memory) is a distinct and interesting problem because its physical address space is distributed across processors while its logical address space is the same as shared main memory. On the top end of this *memory hierarchy*, main memory and backing store are far enough away from the processor that they present one address space for all processors. There is no coherence problem since any processor generating an uncached main-memory address is referring to the same physical word. At the bottom end, registers are so close to the CPU that their physical and logical address are the same, and local. So there is no coherence problem because there is no thought of sharing them. They must be managed directly by the compiler (or programmer). But for cache, a given word can have different, and multiple, locations in a machine.

Caching contrasts with prefetching and static data layout where a given word has only one home, that is, there is only one location in the overall memory of a system where that word is stored. Prefetching tries to hide the cost of accessing that home location, wherever it may be. Static data layout tries to make that home be the best possible place. Caching simply creates as many homes as necessary. Thus, caching addresses the difficulties of the other approaches.

First, consider cache versus static data layout. Since with caching the home is dynamically changing, load balancing is not a problem because the data follows the computation. Similarly, insufficient dependence analysis is not a problem because it does not matter where the data is first allocated, it will move to where it is needed. This is not a panacea because caching only works where there is locality. There is no locality on the first reference to a cache line nor when it has to change homes. Static data layout techniques are still applicable to that class of references.

Second, consider cache versus prefetching. Cache directly addresses the major difficulties with prefetching. The data is brought close to the processor that uses it, and total memory traffic is reduced since most accesses are local. Contention for a particular memory bank is reduced because the data is replicated. Similarly, since the data is already as close as can be arranged, there is no need to look ahead. Increasing distances to main memory makes prefetching more difficult to perform effectively, but caching continues to be effective. Again, where there is no locality, prefetching can be used in conjunction with caching.

There are two reasons that cache is not clearly superior. First, cache is necessarily a request and reply approach. Whichever processor needs data sends the request to the memory which owns the data and that memory responds. Static data layout can work that way as well, but for some kinds of machines (those that are generally thought of as distributed memory instead of distributed shared memory) the owner can send without a request.

The second, and principal, difficulty for cache is that coherence must be maintained: all of the different copies of a variable must represent one consistent picture of the computation. Preserving coherence is expensive. Typically, some mechanism must *invalidate* cache lines so that future references will miss in cache and fetch a valid copy from main memory instead of referencing a *stale* value that would otherwise remain in cache.

Before presenting a formal definition (def. 3.4) of coherence, it is worth considering some of the previous approaches to it. There are three ways coherence has been addressed:

- Hardware – Custom chips that maintain transparent coherence on cache lines.
- VM – Software approaches that use application page faults and operating-system memory-mapping routines to maintain coherence at the page level.
- Compiler assisted – The compiler adds directives that explicitly control coherence, with some hardware assistance.

Which of these three approaches is the most appropriate one depends on the given application and hardware. Furthermore, each of the approaches can be realized in several different ways. Nonetheless, there are important algorithmic similarities between designs in all three categories.

The principal problem of hardware and VM approaches is that they are not truly scalable. For typical ways of scaling memory and problem sizes, the cost of coherence goes up faster than the number of processors. They may be efficient for some particular machine configuration but may prove inefficient for a larger machine of the same design. The difficulty with compiler-assisted techniques is that they are inefficient. By themselves, they are inefficient simply because of the run-time overhead required. This can typically be remedied by hardware assistance, but compiler techniques also suffer from a more fundamental inefficiency: the algorithms cause unnecessary cache evictions (and therefore unnecessary memory traffic) because of the uncertainty of compile-time information.

This thesis explores the different proposed compiler-assisted coherence strategies and develops a framework that puts them in perspective based on their algorithmic nature rather than on their particular implementation (§3.2). The critical distinction is then one of local knowledge versus global knowledge. This framework guides the construction of new coherence algorithms that are scalable while reducing cache evictions to the logically possible minimum for a local strategy. Before developing this framework, this thesis surveys some previous approaches in more detail.

2.2.1 Hardware Approaches

A *hardware approach* is usually part of the memory system, and coherence is maintained transparently, without requiring explicit help from the application or the compiler. The machine appears as if there is just one copy of every word, even to the operating system. This may not be strictly true for a program that violates the consistency model, but we restrict our attention here to correctly synchronized programs [28]. Hardware approaches have been extensively surveyed [46,59,33]. In this section, we review them only in sufficient depth to understand their relation to compiler-assisted approaches.

There are two general approaches to hardware coherence:

- Snoopy – All caches share a common bus and use broadcast (e.g. [31])
- Directories – All locations of a cache line are explicitly recorded (e.g. [33])

There are many intermediate combinations possible as well. Local nodes may snoop a bus for some fixed number of processors and then use directories for the rest of memory (e.g. the Dash [44]). Another trade-off is to fix the size of the directories and to subsequently use broadcast upon overflow. Both approaches have scalability problems.

<u>Snoopy</u>

In a snoopy strategy, all caches monitor a common bus. When a processor writes a value, the write is broadcast on the bus. Any other cache that is holding that value notices the broadcast and updates its local copy (or possibly just invalidates it so that the old value cannot be referenced). Main memory is updated at the same time. When all of the updates are certain to have occurred, the writing processor can continue. This preserves coherence because every copy of a given cache line is guaranteed to be the same. By relaxing the consistency model it is possible to let the writing processor continue before the write is propagated everywhere. In this case, the write must guarantee to complete before appropriate synchronization.

Regardless of the consistency model, the bus is a common medium shared by all of the processors. It serializes the write notices from every processor since only one of them can be broadcasting at a time. When the total write traffic exceeds the bus bandwidth, it will become impossible to utilize more processors. Thus, snoopy systems are unscalable–there is a maximum number of processors that can be used.

Directories

There are several directory strategies [46,59,33]. Previous authors have observed that data frequently read from and written to (as opposed to mostly-read or migratory data) present scalability problems for directory strategies [32].

The simplest directory approach is to keep a bit vector, with one bit for each processor, in addition to the logical cache line in main memory. This simply maps which processors are currently caching a copy of the line. Reads update the map by setting the corresponding bit for the reading processor. Writes scan the map and invalidate now stale copies on every caching processor. The map is updated to reflect that only the writing processor now owns a copy of the cache line.

This preserves coherence because when a new value is written, all other copies are invalidated. Future read references will miss, go to main memory, and find the new value there. There are two scalability problems with this. The first is that the time to write a word is proportional to the number of processors sharing a given word. This can be improved by relaxing the consistency model. In that case, the writing processor can continue immediately. However, the total traffic through the memory system is the same regardless of the consistency model. The second problem is that the memory requirements grow faster than the number of processors. Given the following simple model:

- p – number of processors
- m – data memory per processor
- M – total memory = mp cache lines
- d – directory size = $Mp=mp^2$ bits

Since each logical cache line must have a bit for every processor, total memory requirements ($M+d$) grow faster than useable data memory (M). For a small machine of a given size this may be reasonable. For instance, a 16 processor machine could simply dedicate one word worth of processor map to each logical cache line with no intention of allowing for further expansion. For larger machines, there are several approaches to addressing this problem. Two of the more important ones are:

- Chained-directory schemes – Instead of a map, ownership is represented as a linked list running through memory.
- Limited-directory schemes – The map contains pointers to some fixed number of processors, then broadcast must be used for overflow.

Chained-directory schemes (e.g. SCI [36]) solve the directory overhead problem by allocating a single processor pointer to each logical cache line in main memory. Each logical cache line in actual cache also contains a processor pointer to the next sharing processor. When a value is written, the linked list is followed invalidating every entry along the way. In this approach, the cost of invalidation is proportional to the degree of sharing with high constants. When the degree of sharing is a function of the problem size, instead of being a fixed constant, this makes the cost of coherence per reference O(p). A full map directory is unscalable because it requires O(p) space per word in the program. Chained schemes change that to O(p) time worst case and O(log p) space per word.

Limited-directory schemes allocate some finite number of processor pointers per cache line in main memory. If the degree of sharing is small enough that there are sufficient pointers, this works well. When there are not enough pointers, some compromise must be made to handle the overflow. Three common ways are:

- Broadcast invalidates to all of the processors
- Restrict the degree of sharing
- Coarse maps – A pointer indicates a block of processors

All of these suffer scalability problems. The broadcast strategy is hampered by the same reasons that affect snoopy strategies. Unlike a pure snoopy strategy, migratory data can usually be directly dealt with via the processor pointers in the cache directory entry. The total broadcast load will usually be much less than for a pure snoopy strategy, but still asymptotically bad.

Restricting the degree of sharing will cause unnecessary misses and lower the effective hit rate [12]. A read may invalidate another read-only copy because the number of pointers is exhausted. Widely shared data will continue to miss as the different readers continue to access a given word. In this case, the coherence mechanism is scalable by being defined in such a way as to be inadequate. The actual memory traffic becomes unscalable instead. The degenerate case is that widely shared data must be moved through the memory system every time it is referenced. This has some of the same problems as prefetching;

total bandwidth requirements remain high. Even more importantly, it can have a substantial impact on performance since that latency is no longer being hidden. Keeping the hit rate as high as possible is critical to a good coherence strategy. Compiler-assisted strategies suffer from a low hit rate. However, a hardware strategy that lowers the hit rate to the same level as a compiler-assisted strategy may not be competitive just because of the extra hardware cost. The set of programs for which a limited-directory hardware strategy sacrifice misses and the set for which a compile-assisted strategy sacrifice misses even when their average results are similar. Compiler-assisted approaches handle wide sharing well, but do poorly on apparently migratory data that actually stays on the same processor at run time.

Coarse maps resort to broadcast on a different level. When the processor pointers are exhausted, their meaning is changed—instead of pointing to a given processor, they are understood to point to a block of processors. The size of the block keeps changing in such a way that there are always sufficient pointers. If any processor within a block is caching a given line, then the pointer is set for the whole block. An invalidation then applies to the whole block so that it is sure to invalidate the processor which is actually caching the line. When sharing shows some locality, this works well—for instance, when the machine is multi-processing, all of the sharing for a given cache line will be restricted to a given process which will be restricted to some subset of the total processors. When sharing is wide, but not universal, this degenerates to broadcast and suffers the same drawbacks.

Summary

Hardware approaches to coherence must keep track of every cached copy of a word so that it can be invalidated when a write occurs. Currently proposed hardware schemes require either more than a constant amount of extra space or more than a constant amount of extra time to handle certain program references. While they can be efficient in practice for a given size machine, this ultimately represents an impediment to scalability.

2.2.2 VM Approaches

VM approaches [45, 42] to coherence address the cost of the circuits in hardware approaches by moving the coherence logic into the operating system. Coherence operations are then triggered by VM page faults to pass control to the OS. Since they enforce coherence by changing the status of VM pages, it is necessary that they operate at the page level. There are several advantages to VM-based approaches:

- Flexibility – OS code can be changed and complex protocols implemented
- Cost – No special hardware is needed
- Transparency – As with hardware, user programs still see a coherent machine

The problem with VM approaches is that a whole page is too large of a granularity for most programs, resulting in false sharing. For many useful subscript patterns, the computation can be partitioned so as to avoid false sharing [9]. But in general, false sharing must be addressed. It exists even for programs with regular, analyzable subscripts. There are two basic strategies to VM based coherence:

- Sequential Consistency (§3.1) – there exists only one writeable copy of a page at any given moment
- Release Consistency – multiple writeable copies of a page are permitted

In the sequential consistency case, there can be at most one copy of a modified (*dirty*) page (written in cache but not yet updated to main memory). Since pages are duplicated only when they are read, they all have the same values and all copies are always the same. No two processors see a different value for a variable. If there are multiple writers to a given page it can ping-pong between processors, constantly shifting ownership back and forth. For instance:

```
DO I=1,N
   PDO J=1,N [1]
      A(J)=
```

If `A(1)` through `A(k)` are all on the same page and processors p_1 through p_k split the work of the `J` loop, each taking one iteration, then they are effectively serialized because each has to wait in turn to gain access to the page before it can proceed. That would lead to at least N^2/k changes of ownership. In this particular case, the number of processors sharing each page can be minimized by block distributing the `J` loop. In general, it is more difficult to minimize the sharing by any such simple technique. Even using a block distribution will usually leave a page per processor that is shared by two processors for a default layout. For example consider the case with N=6, p=2, k=2 (for 3 total pages). The

[1] The language-example syntax is explained more fully in Appendix A, but the details should not matter to these introductory examples.

first and third page would be accessed entirely on a single processor, but the second page, containing `A(3)` and `A(4)`, would bounce back and forth 2*N times in the above loop. Careful layout is needed to avoid this problem, requiring analysis similar to the static layout case. Without the ping-pong effect, that could be easily reduced to 0 when the `J` loop schedule is the same every time because if p_1 gets `A(3)` on the first iteration of the `I` loop, it will find it in cache for every subsequent iteration. In short, the ping-pong effect can be a serious impediment to performance and has to be addressed [9].

One solution to this is simply to remove the constraint that only one copy of a dirty page can exist at one time. That also removes the constraint that more than one copy of the same dirty word can exist at one time. Allowing that to occur would violate program semantics. To maintain correctness, the system assures that dirty words are updated at release synchronization points and the programmer assures that synchronization is correct (i.e. the program will not write the same word on two different processors at the 'same' time). Consider for instance the following code fragment:

```
FORK
A(pid)=
BARRIER
A(3-pid)=
JOIN
```

where there are two processors with `pid`=1 and `pid`=2, respectively. If caches operated without any coherence, both processors would have new values of `A(1)` and `A(2)` at the `JOIN` and those values would disagree between the processors. Some part of the release consistent system (e.g. write buffers which empty before synchronization) causes both new values to be written back to main memory at the `BARRIER`, which implies a release, and then those values are updated (or invalidated) in all other caches that contain a copy.

That solves the ping-pong effect problem because the page never changes ownership. Each processor just does what it needs to and informs the others of the results later. However, the problem now is to propagate the appropriate information. A given page can be partially modified on one processor and partially modified in a different fashion elsewhere. Those individual writes must have their effects merged and not merely propagated on a whole page basis.

KSR [40]

The KSR is a cache-only memory architecture (COMA) machine: a given main memory page does not have any particular home but can be owned by different processors over the course of the computation. This operates like cache by exploiting locality to move data closer to the processor that will need it. These are still main memory pages and therefore several cycles away from the processor. But main memory on the same node is much closer than main memory on a separate node. Since whole main memory pages are being moved, there is no need for any backing store as there is in the case of typical processor caches that move only a cache-line's worth of data out of a page. Pages are usually moved only for coherence, i.e. when logically necessary, and not merely to bring them closer (at the cost of evicting other pages to be referenced on the same processor). There may additionally be hardware caches, but they are trivially kept consistent with the local memory.

The KSR implements sequential consistency. This conveniently solves the problem of merging updates, which would be problematic for release consistency when there is no backing store. Before a write can complete to a page, the processor making the write must own the page locally, by moving it if necessary.

The difficulty with this approach is that sequential consistency imposes a high penalty on large pages because of the ping-pong effect. This could be helped with a smaller page size, but then the cost of invalidation would become problematic for the same reasons as in pure hardware cases. It could also be helped with careful layout, but the problems are then similar (but not identical) to those of the static data layout case (for instance, sends and receives could be modeled as writes and reads to reserved pages which the coherence protocol then moves but each ‘real’ data page is guaranteed to only be accessed by a single processor).

COMA machines (and other machines that can operate that way) are interesting because they expose all inter-epoch reuse (§3.1.1). If a coherence protocol can work effectively at this level (of main memory pages versus network memory instead of at the cache line level), then it will be effective in capturing inter-epoch reuse which might otherwise be lost because of small caches.

Lazy Release Consistency [39]

Lazy Release Consistency (LRC) is a good representative of a class of release consistent coherence algorithms intended for distributed shared-memory machines. Without going into the details of consistency models (§3.1), the important aspect of LRC is that it tackles the merge problem directly by maintaining *diffs*, sets of changes that must be propagated and merged at synchronization points. This directly solves the ping-pong problem by allowing multiple writers and it keeps network traffic low by sending changes to processors only when there is good reason to think they are needed, instead of broadcasting to every processor.

LRC has substantial overhead and therefore seems suitable only for operating on memory pages and not cache lines. It addresses scalability concerns well by avoiding directories and immediate broadcast. LRC piggybacks data *diffs* on top of already obligatory synchronization messages and may in many cases impose little extra message overhead for machines that have substantial message startup costs. When it does impose extra overhead, that extra data will be propagated to every processor as they synchronize regardless of where it is really needed. That represents an impediment to scaling similar to chained-directory schemes; the total coherence traffic increases with the problem size *and* the number of processors.

Kontothanassis & Scott (K&S) [42]

The K&S approach is probably one of the best for handling the caching of main memory data in a small, local cache, as is done on a shared-memory multiprocessor (as opposed to caching network pages in local main memory, as is done on a distributed-memory multiprocessor). It adopts the release-consistent approach. The actual merging is done by the hardware to main memory only, which is updated with each word written. Other sharing processors have their pages invalidated under software control at acquire points. An important scalability feature of the K&S approach is that invalidation needs only to scan the shared pages that a given processor has mapped in and not all of the shared pages in the program.

On a small test suite, this thesis measures the necessary costs incurred by the K&S approach and contrasts them with the costs for the methods proposed in this thesis (chapter 9). The essential aspects of K&S for that purpose are summarized in Appendix C.

The use of release consistency solves the false sharing problem between barriers, i.e. it removes the ping-pong effect. However, false sharing across barriers remains a problem for the K&S approach. A read-only word that falls on a weak page because of other writes to the same page is also invalidated at a barrier even though it does not need to be.

Smaller page sizes reduce the impact of this effect. But, smaller page sizes incur higher cost because as the number of pages grows so does the cost of *weaklist* maintenance. This thesis examines the trade-off of these effects on total global memory traffic (chapter 9).

Chapter 3

Cache Coherence Model

Compiler-assisted strategies are usually applicable to *epoch*-based parallelism, a simple synchronization model consisting only of barriers. Within this model, coherence has a simple characterization based on the type of access and the number of barriers crossed. This in turn implies a consistency model. We detail these and then use them to develop a framework for different coherence strategies.

3.1 Cache Model

3.1.1 Machine and Execution Model

Cache coherence schemes are intended for shared memory multiprocessors with processor local caches. By shared memory we mean that any variable can be accessed by every processor in the same way and that no cooperation is needed between processors to effect this. The physical memory can be equidistant from all processors or local to particular ones.

By cache, we mean the usual notion of a hardware cache that holds some subset of previously referenced values and the program executes without concern for managing it. In particular, the cache is not just a local memory that the program is burdened with precise management of. Misses and evictions are handled transparently by the hardware. This will be relaxed slightly to make software coherence achievable. Caches have some *invalidate* mechanism, either implicit or explicit, which causes certain cache lines to no longer be valid so that subsequent references will cause cache misses and read from main memory. In some cases, the coherence strategy will also be concerned about when to *update*, moving dirty values from cache to main memory. As instructions, these will be referred to as INV and UPD, respectively.

3.1.2 Scheduling and `DOALL` Semantics

The tasking model is fork-join style parallelism. The scheduling of threads to processors at a fork is not known at compile-time. This facet of the problem, perhaps more than any other, distinguishes coherence methods from distributed memory techniques, such as those used in the Fortran D compiler [35].

The iterations of a parallel loop are scheduled on processors at run-time. A common reason for this is to achieve load balance. This is a significant advantage that shared-memory systems (as commonly understood) have over distributed memory systems. The compiler must assume that a given iteration could be executed on any processor. All coherence strategies known to us make this assumption.

Parallel loops with no internal synchronization, e.g. a Fortran `PDO (DOALL)`, are sufficient to guarantee that iterations can be scheduled in any order as needed for load balancing. Such `PDO`s make a very strong guarantee: there are no race conditions, i.e. if a variable is written in one iteration of a parallel loop, no other iteration accesses that variable. This does not preclude multiple readers.

<u>Epochs</u>

Fork-join programs are composed of a series of *epochs*[15]. Each epoch is either a (fork-join) parallel loop with no internal synchronization, e.g. a Fortran `PDO (DOALL)`, or a serial region between parallel loops. Serial regions can be nested serial loops and/or those parts of serial loops enclosing parallel loops. The following example has m+2 epochs:

$$\mathit{Epoch}\left\{\begin{array}{l}\text{PDO I} = 1,\text{N}\\ \quad \text{A(I)} = \text{I}\\ \text{END DO}\end{array}\right.$$

$$\mathit{Epoch}\left\{\begin{array}{l}\text{B}(1,1) = 0\\ \text{DO J} = 1,\text{M}\\ \quad \text{DO I} = 2,\text{N}\\ \quad\quad \text{B(I,J)} = \text{B(I}-1,1) + 1\\ \quad \text{END DO}\\ \text{END DO}\end{array}\right.$$

$$\begin{array}{ll} & \text{DO J} = 1,\text{M}\\ \mathit{M\ Epochs}\left\{\begin{array}{l}\\ \\ \\ \end{array}\right. & \begin{array}{l}\quad \text{PDO I} = 1,\text{N}\\ \quad\quad \text{C(I,J)} = \text{A(I)} + \text{B(I,J)}\\ \quad \text{END DO}\end{array}\\ & \text{END DO}\end{array}$$

The first epoch is simply a single parallel loop. The second epoch is the serial loop nest and the single assignment before it, i.e. all of the serial work before the next parallel loop. The final loop represents one epoch for each time the inner parallel loop is entered. The loop control instructions for the final `J` serial loop will be in a (small) serial epoch but

```
PDO I=1,2
   A(I)=5
END DO
PDO I=1,2
   A(3-I)=
      A(3-I)+2
END DO
PDO I=1,2
   B(I)=A(I)
END DO
```

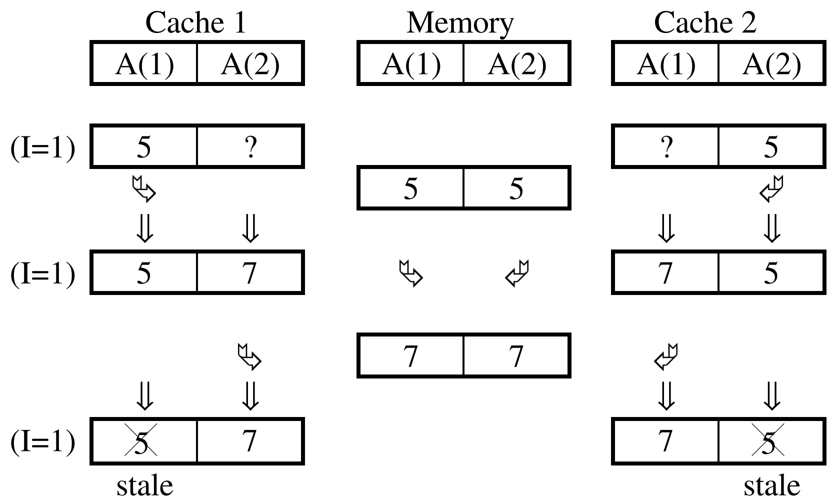


Figure 3.1: Example of Stale Access in the Absence of Coherence Control

its location depends on the actual implementation. In practice, these tiny epochs will often be ignored since they have no impact on shared variables. All code executed between two synchronization operations (either fork to join or join to fork) is an epoch.

3.1.3 Coherence

The problem this work addresses is how to manage caches so that they maintain a consistent view of main memory. Values written to cache must be *updated* before they can be accessed by some other processor. Values in cache must be *invalidated* at an appropriate time to make sure that new values are brought in from main memory when needed.

Figure 3.1 gives an example of the coherence problem. If the program is run on two processors, each with its own cache, but with a shared memory, stale values will be referenced in the third loop. Assume that processor p_i gets iteration i on all loops. On the first loop, p_1's cache will have '5' for the value of `A(1)`. Similarly, p_2's cache has `A(2)` with the value '5'. Either at the time of the write or before the end of the loop both of these values will be written back to main memory. In the second loop, both of the previous values stay in the their respective caches. Now, p_1 assigns '7' to `A(2)` and similarly for p_2. The problem arises in the third loop. The '7' that p_2 wrote to `A(1)` is available to p_2 (obviously) and is sent to main memory, but p_1 never became aware of that change and the old value of '5' is still in its cache. That value is *stale*. The purpose of a coherence protocol is to prevent stale values from being accessed. In this case, `A(1)` needs to be invalidated in p_1's cache before the third epoch begins so that the reference to it will miss and the new value of '7' will be brought in from main memory.

This example shows the essential features of staleness. First, for a value to be stale, there must be three accesses in three different epochs. If there are only two accesses, the second one will either be a hit on a valid value or it will be a miss and the valid value will be fetched from main memory (other possibilities are excluded by the model being restricted to data-race free loops). For Example, the read of `A(2)` on p_1 in the second epoch is valid even though a write in a different epoch preceded it. Second, the second reference must be a write. If it were a read, there would be no new value that needed to be communicated and the remaining access could not then be stale. Third, the final reference must be a read, obviously. These criteria together form an *access triple*:

Definition 3.1: *Access Triple* (AT)

Three references to the same variable, e.g. `A`, all in different epochs such that:

- A_{ac} - the first one in (run-time order) is any access to `A` (the *essential access*)
- A_w - the second one is a write to `A` (the *essential write*).
- A_r - the third one is an upwardly exposed read of `A` (the *essential read*).
- The complete AT, A_{at}, is the triple (A_{ac},A_w,A_r).
- The reuse arc (RA), A_{ra}, is just first two components, (A_{ac},A_w), of the AT.

In any of the epochs there can be additional references of any type, but these three must be present to form an AT. ATs are separable for each variable. Accesses to `B` have no effect on finding ATs for `A`. Even for a single variable, since the three parts of an AT must all be in different epochs, the reference pattern inside a particular epoch is not relevant to finding ATs. References inside an epoch can be analytically compressed into a single statement for purposes of finding ATs (see Appendix A for this and other special notation). Keeping references distinct is still relevant for placement of coherence control in some strategies however. Note that the definition does not specify the smallest such AT possible, but only that the pattern exist. ATs for the same variable can overlap in various ways including entirely containing one another.

Coherence matters only for the first reference in a given epoch. Once the first reference is coherent, subsequent ones must also be since the absence of data races guarantees no writes could make them stale. For example,

```
PDO I
   A(I)=
   =A(I)      This is not part of an AT, regardless
   =B(I)      This either is or will be made coherent
   =B(I)      This is not part of an AT either
ENDDO
```

This observation allows every epoch to be represented as a single reference for each variable. A variable is either read, written, or both in a given epoch. That summarizes all of the useful information. It does not matter how many times it is referenced. The notation reflects this by having each reference symbol, e.g. A_{ac}, represent a different, whole epoch.

ATs will be symbolically represented by  over an *epoch graph*. An epoch graph is the control flow graph where every epoch is reduced to a single node. The AT symbol then touches three epochs, the three that compose the given AT. Figure 6.1 is a simple example and figure 6.2 is a slightly more complex example with a simple inter-epoch `IF`.

Not every AT produces a stale value. If A_w and A_r are on the same processor, A_r will necessarily get the value from A_w and the stale value from A_{ac} will not be a problem (e.g. if, in figure 3.1, p_1 had executed iteration 2 in the second epoch, there would not be staleness). If A_{ac} and A_r are on different processors, A_r will not be able to get the old value and will be valid whether or not A_w occurred on the same processor (e.g. if, in figure 3.1, p_1 had executed iteration 2 on epoch 1, there would be no staleness). *Staleness* can now be precisely defined:

Definition 3.2: Run-time *Stale* value

The last value of an AT, e.g. A_r in $(A_{ac},A_{w,}A_r)$, such that:
A_{ac} and A_r occur on p_i
A_w occurs on $p_{j,i\neq j}$, i.e. a different processor than A_r.
i.e. this pattern occurs: $A_{ac}[i]\ (A_{ac}[\neg i])^*\ A_w[\neg i]\ (A_r[\neg i])^*\ A_r[i]$
(from Karlovsky [38], notation summarized in Appendix A)

$(A_{ac}[\neg i])^*$ and $(A_r[\neg i])^*$ represent other references that could occur without preventing the essential AT pattern from existing. If the penultimate term, $(A_r[\neg i])^*$, were to include $A_{ac}[i]$, it would mean that the last read were not actually stale. If it were to include

$\mathtt{A}_w[\neg i]$, it would not change anything. It would merely parse the expression a different way.

Previous authors have shown how to detect this condition at compile-time [15,21,38]. Conservative assumptions must be made. Any processor schedule could be used and any control flow path could be taken. The first of these implies that the distinction between $\mathtt{A}_{ac}[i]$ and $\mathtt{A}_{ac}[\neg i]$ cannot be made at compile time. Any $\mathtt{A}_{ac}$ must be assumed to be both, etc. There is one sometimes useful exception: the same processor always executes the serial section. We will we refer to this as processor 0. We use '≥0' to mean any processor taking part in a `PDO`, including possibly processor 0 where the compiler cannot know anything specific about it. What can actually be detected at compile-time is then:

Definition 3.3: Compile-Time *Stale* value

The last read in the pattern:

$\mathtt{A}_{ac}{}^{+}\ \mathtt{A}_w\ \mathtt{A}_r{}^{+}$

or more precisely:

$(\ \mathtt{A}_{ac}[0]\ \ (\mathtt{A}_{ac}[\geq 0])^{*}\ \mathtt{A}_w[\geq 0]\ (\mathtt{A}_r[\geq 0])^{*}\ \mathtt{A}_r[0]\ \ )\ |$

$(\ \mathtt{A}_{ac}[\geq 0]\ \ \mathtt{A}_{ac}{}^{*}\ \mathtt{A}_w\ \mathtt{A}_r{}^{*}\ \mathtt{A}_r[\geq 0]\ \ )$

This strictly subsumes definition 3.2. It counts every AT as stale that the compiler cannot disprove will be run-time stale. It counts many reads as stale that will not actually be so at run-time but misses none. But for the question of serial sections, this means every AT defines a compile-time stale value (though not necessarily a unique one). The converse, that every stale-value (run-time or compile-time) is part of an AT, is obvious from the definition.

The second conservative assumption, that all control flow paths must be considered, means that a particular essential read might be at the end of two different ATs, one of which makes it stale and one of which does not. In that case, it is considered to be compile-time stale.

The central issue of this thesis can now be defined:

Definition 3.4: *Coherence*

Invalidating the data specified in ATs such that no run-time stale values are referenced.

This only defines what is needed for correctness. Invalidation is used to assure that no stale value will be in cache when it is referenced. Instead, a cache miss will occur and the new (valid) value will be brought in from main memory.

This is a conservative definition which says only that all stale values must be handled for correctness. It says nothing about how, when, or at what cost. Within this constraint, coherence strategies strive for a good balance of implementation cost and execution time. One important trade-off is where the invalidate actually occurs. It can occur synchronously with the essential write of the AT (generally under hardware control). It can occur any time between the essential access and the essential read (generally under software control). Another important trade-off is the cost of the invalidate itself versus the cost of extra cache misses for a cheaper, but imprecise, invalidate.

The strategies developed in this thesis strive for efficient coherence by grouping the invalidation work and using fast invalidates at the cost of inducing some logically unnecessary cache misses.

3.1.4 Consistency

The programming model places certain limits on what kinds of coherence are possible and what the underlying system consistency model must look like. Programmers generally view memory as being *sequentially consistent* (SC), defined by Lamport [43] as:

> [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The second half of this definition is *program order*. Whatever happens on this processor must take effect in the same order as specified by the program, regardless of how the effects appear to other processors. The problem is that SC is too strong of a model for efficient implementation. Numerous alternative consistency models have been proposed [2,39] that relax the first half of the above definition, while maintaining program order, so that accesses are guaranteed to have observed orders between processors only with respect to synchronization. They are guaranteed to have this behavior for programs running in a protected process space and not necessarily for those running in privileged, operating system modes. We will refer to these generically as *release consistency* (RC) models in this thesis. Aside from the matter of writing synchronization code, all of these relaxed models look like SC to the programmer, for data-race-free programs [2].

For `DOALL` programs, the nature of the synchronization is precisely specified and not directly coded. Because of this and since `DOALL` semantics (§3.1.2) specify no data-races, the consistency model can be any of these. Additionally, compiler-based coherence strategies can assume 'no consistency' as the model, i.e. neither the hardware nor the operating system guarantee anything about the apparent order of accesses between processors (other than local program order must be respected).

Invalidates fit easily in this model. They occur on the same processor as the references they are for. Program order guarantees they are performed as expected. However, updates are intended to have an effect on another processor and must rely on some lower level consistency protocol to guarantee that their effects will be propagated. We assume that the underlying system is RC with respect to main memory, for efficiency. This is not the same as being RC in general which means being consistent with respect to every other processor/cache. Being RC with respect to main memory is a considerably weaker model.

To understand the impact on updates, consider the following simple situation:

```
PDO I=1,N
   A¹(I)=           — actually occurs on p_w
END DO
PDO
   ...
END DO
PDO I=1,N
   =A²(I)           — actually occurs on p_r
ENDDO
```

Because scheduling is unknown, there is no way to know if the read and write of `A` will occur on the same processor. The actual written value must be updated to main memory *before* e_3 begins. Even if p_w knew something about when p_r would actually read the value, it could not delay the update into e_3 because the underlying RC semantics would not guarantee the value would reach main memory before p_r would fetch it. The update could be delayed from e_1 until e_2, when another barrier would guarantee the update happened. But the actual movement must still occur and not much is gained. Therefore, the standard assumption for compiler-based coherence strategies, including the work in this thesis, is that the update happens in the same epoch as the write. It may happen implicitly with a write-through cache that buffers writes and guarantees they complete before a synchronization completes. Or, it may happen with an explicit update that flushes dirty values from a write-back cache.

A similar analysis applies for reads that will miss. They cannot occur before the synchronization that begins their epoch, even though the invalidate may before that point. In the absence of prefetching, this is not even an issue since memory reads will not occur before they are requested in the program.

If something is known about the parallel loop schedules, the picture is considerably different. It is no longer always necessary to update dirty values because it may be provable they will be referenced on the same processor again in subsequent epochs. These techniques are more the province of distributed memory compilation techniques and not applicable to the unknown schedule case.

Other programming models are possible, but for the one being considered in this thesis, shared memory and unknown parallel loop schedules, the consistency model follows naturally. It is:

- Accesses are RC with respect to main memory, not other processors.
- Writes are performed before subsequent synchronization.
- Reads are performed after preceding synchronization.

3.1.1 Inter-Epoch Reuse

One of the objectives for a coherence strategy is to preserve as many cache hits as possible. From a coherence point of view there are two distinct types of cache hits, *intra-epoch* and *inter-epoch*. For example, in:

```
PDO I=1,N
   DO J=1,M
      =B(I,J)*A¹(I)
   END DO
END DO
PDO I=1,N
   =A²(I)
ENDDO
```

Let p_1 get iteration 1 of both `I` loops. `A(1)` will be accessed `M` times on p_1 in the first epoch, e_1. For a sufficiently large cache, `M`-1 of those accesses will be intra-epoch hits, found while p_1 is still executing e_1. In e_2, p_1 will reference $\mathtt{A}^2\mathtt{(1)}$ and, for a sufficiently large cache, find it still in cache from e_1:$\mathtt{A}^1$, producing an inter-epoch hit.

The distinction matters because preserving intra-epoch reuse is logically a trivial problem. Coherence and intra-epoch reuse can both be maintained by simply invalidating all shared data at epoch boundaries. In most cases, there is an equally simple implementation: change the page map entries at epoch boundaries. This prevents any shared data from crossing epoch boundaries and thus surely preserves coherence since it subsumes every AT.

Thus, coherence strategies are principally concerned with how to preserve inter-epoch reuse while maintaining coherence (some approaches have incidental negative effects on intra-epoch reuse, [47,chapter 5]). All coherence strategies that are concerned with release consistent memory models make the assumption that this is worth pursuing [15,13,50,42,47]. Conventional hardware approaches often achieve sequential consistency instead of merely release consistency. In that case, it cannot be clearly said they depend on the existence of inter-epoch reuse to justify their implementations. However, some hardware coherence systems explicitly preserve only release consistency [44] and also rely on the existence of inter-epoch reuse. Regardless, release consistency is essential for obtaining high performance from large scale multiprocessors [27,63] and all coherence strategies must eventually justify their existence on finding inter-epoch reuse.

Despite this, it is not clear how much inter-epoch reuse real programs have because of finite cache size. Such reuse exists for sufficiently large caches (chapter 9). However, smaller caches will tend to sacrifice inter-epoch reuse before intra-epoch reuse. How much survives is a question about how large of an application people try to run, how many processors they apply to the problem, and how cache is added to each processor. Cur-

Program	Significance Level* 5%	100%	Class A size / "sample" size
	K cache words		
BT	16	128	20
LU	8	128	25
SP	8	16	30
IS	32	64	128
CG	4	128	20
EP	128	128	16
FT	<1	<1	32
MG	8	128	512
Ocean	1	8	n/a

* (Inter-epoch reuse)/misses > significance level (for caches≥ this size)

Table 3.1: NAS Inter-Epoch Reuse

rently, the unsettled state of parallel programming makes it difficult to measure the average characteristics of a 'real' parallel program.

Previous authors in the area have found significant inter-epoch reuse on their limited cache test suites [14]. Recent work by McIntosh [48] has found significant inter-*loop* reuse on the "sample" data size, standard NAS benchmark [8] and Ocean (from the PERFECT benchmark [20]) for sequential versions of those codes. His notion of inter-loop is between outer loops as recognized by intra-procedural compiler analysis, not simply between inner-most loops. In several of the NAS applications, this corresponds exactly to the loop that would be parallelized and in others it is more pessimistic (lower reuse) than inter-epoch reuse because the parallel loops are not the outer ones. We summarize his results that are relevant to this thesis (table 3.1) . All of the codes except EP showed significant inter-epoch reuse at reasonable cache sizes, for instance LU with an 8K word cache had approximately 20 times as many misses as it did inter-epoch hits. Three of the codes, SP, FT, and Ocean, would have double the miss rate on reasonable cache sizes if inter-epoch reuse were lost.

The final column is how much larger the "Class A" [7] size problems are than the "sample" size problems, e.g. a real instance of FT would take 32 times as much memory as the size for which the tests were run. Problems this large were not attempted by McIntosh nor are they accessible in our simulator. However, the problem size serves as a rough guide as to how many processors need be applied to a parallel instance of the problem to preserve the inter-epoch reuse. For instance, one would expect that applying 20 processors, each with a 4K word cache, to CG would still show inter-epoch reuse to be 5% of the total misses. Only MG requires an unreasonably large number of processors (since the "Class A" size is 256 rows with the total data size 256^3) to preserve inter-epoch reuse.

When a code does not have significant inter-epoch reuse on a given machine, there are several ways to realize greater inter-epoch reuse:

- Bigger cache
- More processors
- Using all-cache machines
- Program transformation

The first approach is the obvious one. It merely says that if coherence techniques are available to exploit the remaining inter-epoch reuse, it is simply necessary to buy more cache to realize that reuse. The second approach, more processors, is a variant on the first. For parallel machines, the number of processors that can be applied to a problem is one of the variables. Using more processors means more total cache. Often, it will make sense to use those extra processors because the total problem solution will become more efficient when more of it fits in cache.

The third approach gains more cache by changing the definition of cache. Instead of having cache be a small amount of memory closer to the processor than main memory, it treats all of memory like cache in the sense of moving the home location of a page to where it is referenced [40]. This is not a solution to insufficient cache size because such machines still have cache at a lower level than the attraction memory which is being kept coherent. However, it is a type of a machine where coherence strategies become relevant since inter-epoch reuse, in the main memory sense, is relevant.

Instead of applying more hardware to recover the reuse, some codes can be restructured so that they have more reuse for a given cache size. This has been extensively studied for the uni-processor case [10,62]. Parallel loops do not change the nature of such approaches nor their applicability. However, the relation between inter-epoch and intra-epoch reuse can be changed. Consider the following loop with P total processors and C words of cache:

```
PDO I=1,N
   DO J=1,N
      =A(J)
```

If N>C, the reuse from `A` to `A` on subsequent iterations of the `I` loop will be lost (assuming P>N). Unroll-and-jam could be applied to preserve this reuse by converting the loop to:

```
PDO I1=1,N,B
   DO J=1,N
      DO I2=I1,I1+B-1
         =A(J)
```

For B iterations of the I2 loop, reuse will be preserved on A. For examples like this, where the parallelism is in the outermost loop, unroll-and-jam can do nothing for inter-epoch reuse (parallelizing the I2 loop would defeat the purpose). However, for those cases where the parallelism is trapped in the inner loop of an unroll-and-jam pair, the reuse which is recovered is necessarily of an inter-epoch sort. Consider:

```
DO I=1,N
   PDO J=1,N
      A(I,J)=A(I-1,J-1)+A(I-1,J)
```

The outer I loop carries the dependence and the loops cannot be interchanged with J remaining parallel. However, if N>C, let B=C*P, and the loops could be transformed to:

```
DO J1=1,N,B
   DO I=1,N
      PDO J2=J1,J1+B-1        — block scheduled
         A(I,J2)=A(I-1,J2-1)+A(I-1,J2)
```

The amount of data referenced on one iteration of the I loop is B words, or just enough to fill the cache on each processor. Therefore, there is reuse in the next iteration of the I loop. This reuse cannot be converted to intra-epoch reuse and still preserve all of the parallelism. For this particular loop structure, the J1 loop must stay serial to carry the dependence. More aggressive restructuring such as wave-fronting the loop would produce similar results leaving the new reuse as inter-epoch.

Many programs have inter-epoch reuse available with reasonable cache sizes. Programs with larger cache requirements will have inter-epoch reuse exposed as cache costs decrease, the degree of parallelism is increased, and program transformations are applied. For these two cases, coherence strategies that preserve inter-epoch reuse are worth exploring. Finally, programs with the largest of cache requirements will always prevent inter-epoch reuse from being found because of cache size limitations . For such programs, coherence strategies will matter only as they are applied to all-cache machines.

In this thesis, we are not concerned with how to generate more inter-epoch reuse, rather we explore how to preserve the available reuse.

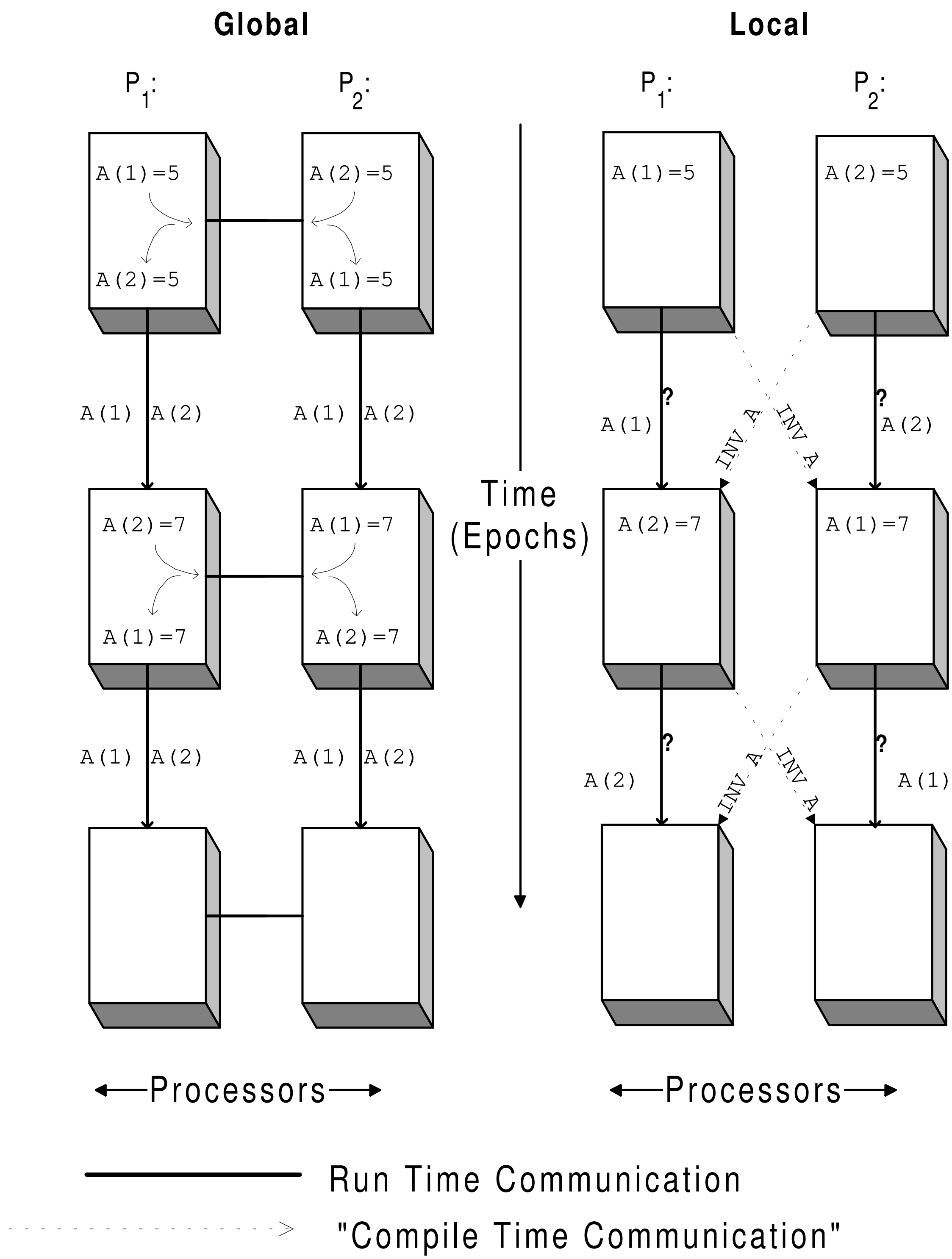


Figure 3.2: Global versus Local Coherence

3.2 A Framework for Coherence

All proposed and existing coherence strategies implement coherence by focusing on the essential write of an AT. Before the end of a given epoch, a processor is responsible for invalidating everything in its cache that some other processor could have written in the same epoch. Looking at figure 3.1, this would mean that p_1 was responsible for invalidating `A(1)` (written on p_2) before the end of epoch 2. This directly addresses definition 3.2. An essential part of staleness is: $A_w[\neg i]$ $A_r[i]$ (a write epoch followed by a read epoch). When p_i invalidates everything in $A_w[\neg i]$ before the $A_r[i]$ epoch, coherence is preserved by breaking the AT pattern. What p_i knows about $A_w[\neg i]$ is the principal factor determining the nature of a coherence strategy. Making an estimate which supersets the actual dependence pattern can be much cheaper than an exact determination.

3.2.1 Local versus Global Knowledge Coherence Strategies

The first important distinction is whether or not global information is communicated at run-time to p_i. If it is, a *global* coherence strategy is being used. If no global information is communicated at run-time, a *local* strategy is being used. Global strategies are usually thought of as hardware strategies, e.g. snoopy caches [55,58] and directory based caches [11,44,61]. Other global strategies, e.g. OS level page strategies [56], are software strategies.

Figure 3.2 schematically represents the difference. The effects of a global strategy are shown on the left and those of a local strategy on the right (for the code in figure 3.1). In the global strategy, as soon as a value is changed, the information is communicated to other processors (whether by invalidate or update protocol does not immediately matter). When p_1 writes `A(1)`, that fact is sent directly to p_2 in the same epoch. At the end of each epoch, everything stays in cache undisturbed. For the local coherence case, the information that would have been exchanged at run-time is approximated at compile-time. The dashed arrow represents the information that any part of `A` could have been written in epoch *i* and that must be 'communicated'. Therefore, epoch *i*+1 begins with an invalidate of `A`. The same-processor, inter-epoch arrows represent *some* information staying in the cache. What exactly remains depends on the particular local strategy being used.

Cache misses have four causes: initial loading, cache size, cache organization (e.g. associativity), and invalidation to preserve coherence between processors (sharing induced). Coherence strategies are concerned only with the last category, sharing induced misses.

No currently existing global strategy causes a logically unnecessary sharing miss (ignoring, for the moment, false sharing). They are, in that sense, optimal. The drawback of global strategies is that scalability is impaired by the cost of maintaining global knowledge at run-time.

If no global knowledge is shared at run-time, then coherence must rely on locally collected knowledge plus whatever global knowledge was collected at compile-time. Previous local knowledge strategies have been referred to in the literature as software [15,21] or hardware [50] strategies depending on whether most of the work was done in software or hardware. We consider all of these strategies to be similar and refer to them collectively as local strategies.

A local strategy will likely never result in an optimal hit ratio for a processor because some useful run-time knowledge will be unavailable. The effectiveness of local strategies can vary widely depending on the program being run and the strategy being used. The principal advantage of local strategies is that they are scalable because no global knowledge need be communicated at run-time to maintain coherence. They rely on what could happen, not what does. We use 'could happen' to mean that the compiler cannot disprove it.

```
PDO I=
   A(I)=
ENDDO
PDO I=
   IF (false) A(I)=
ENDDO
PDO I=
   B(I)=A(I)+1
ENDDO
```

Figure 3.3: Global versus Local, hit rate improvement

Figure 3.3 gives an example of where a local strategy would fail to achieve the same hit ratio as a global strategy. Since the condition is the second epoch is unanalyzable at compile-time, all potential reuse of cached values of A between the first and third epochs will be lost using a local strategy but preserved using a global strategy.

3.2.2 Schedules Disproving Staleness

Before discussing finer divisions of coherence strategies, an important observation needs to be made:

Theorem 3.5, Local Schedule Theorem: A referenced value is never stale in the immediately subsequent epoch on the same processor.

Proof: The pattern in definition 3.2 cannot be satisfied because there can be no epoch containing $A_w[\neg i]$ between $A_{ac}[i]$ and $A_r[i]$.

This has important practical ramifications. Consider the following code fragment:

```
PDO I
   = A¹(I)                     e0 (epoch 0)
ENDDO
PDO I=1,N
   A²(f(I)) = A³(g(I))+1  e1
ENDDO
PDO I
   = A⁴(I)                     e2
ENDDO
```

The PDOs guarantee no race conditions (§3.1.2). When $A^4(I)$ (for a given I) is read in e_2 there are four cases to consider:

1) It misses in cache and gets the new value from main memory.
2) The value found in cache was written at A^2 in e_1.
3) The value found in cache was read at A^3 in e_1.
4) The value found in cache was not referenced in e_1.

Case 1 works correctly. In case 2, the A_r^4 in e_2 gets the new value, A_w^2, written in e_1, regardless of what else has happened. Case 3 is the interesting one. Since the same A(I) was read in e_1, it could not have been written in e_1 (except on the same processor). Therefore A_w^2 in e_1 could not make A_r^4 in e_2 stale. Some write before e_1 might make it stale but in that case A_r^3 in e_1 would also be stale. Assuming A_r^3 in e_1 is not stale, A_r^4 in e_2 cannot be either. Case 4 is stale if A could have been previously referenced and it must be handled by whatever mechanism is being used.

By itself, the compiler cannot make much use of this because any reference in e_2 is compile-time stale and might be run-time stale. This is not merely an artifact of a poor definition of compile-time stale but reflects the worst case scenario where the reference pattern on e_1 is completely different than on e_0 and e_2 , leaving references in e_2 run-time stale. For example, invalidating before e_2 is essential if the reference pattern were:

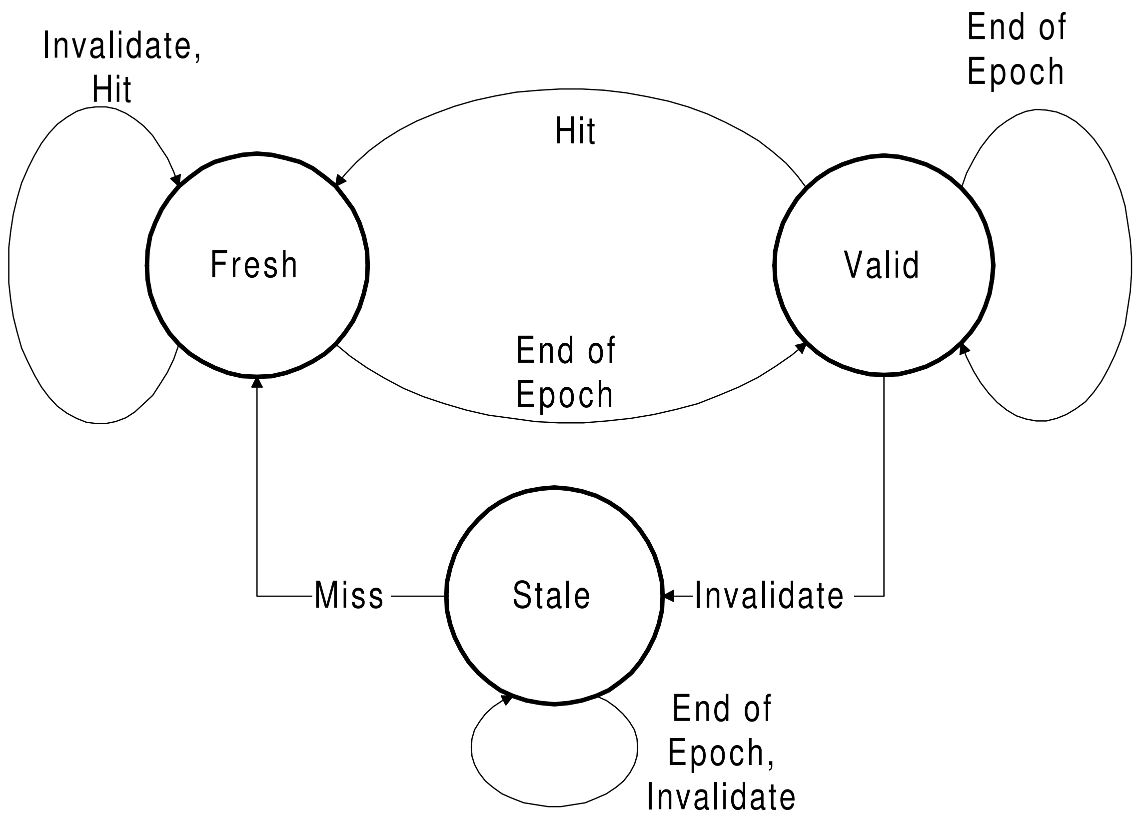


Figure 3.4: *Fresh* cache state

<u>Epoch</u>	<u>Reference</u>	
0	`A(I)`[I]	`A(1)` is referenced on processor 1, etc.
1	`A(I)`[N-I+1]	*f* & *g* cause the reference pattern to be 'backward'
2	`A(I)`[I]	w/o invalidate would use `A(I)` from e_0

The statement of theorem 3.5 essentially specifies the mechanism for gaining advantage from this situation: never invalidate anything just referenced in the immediately preceding epoch (implementation discussion is deferred to chapter 4). It is useful to view this as another cache state (figure 3.4).

Definition 3.6: *Fresh* Cache Line

A cache line is *fresh* during an epoch in which it has been referenced on the processor that referenced it.

This can have a tremendous impact on hit rates. Consider what happens when a static strategy is applied to:

```
DO I
   PDO J
      A(J)=A(J)+1
   ENDDO
ENDDO
```

The read of `A(J)` is stale from previous iterations and must be invalidated. Thus the read of `A(J)` will never hit under a static strategy. If theorem 3.5 is applied and the schedule for the `J` loop is the same for every iteration of the `I` loop, then the read of `A(J)` will hit every `I`>1 iteration, assuming suitable cache size and organization.

3.2.3 Local Dynamic versus Local Static Coherence Strategies

Local strategies can be either static or dynamic. Static local strategies decide which cache lines to invalidate and when that invalidation should occur, using only compile-time knowledge. In contrast, dynamic strategies can use *local* run-time information as well. This comes in two forms, knowing the local processor schedule and the actual execution path of the program. Implementing theorem 3.5 is the usual way of utilizing this knowledge. Other methods are conceivable.

Static strategies cannot utilize theorem 3.5. Since no run-time information is available and worst case scheduling is assumed, it is impossible to decide if a value is on the "same processor". Static strategies preserve coherence for all possible intra-epoch control flow paths and all possible schedules, then apply that set of invalidates to all programs. Static strategies can still exploit some inter-epoch reuse however.

Dynamic strategies strictly improve on static strategies. Any static strategy can be changed to a dynamic strategy that will not cause any more coherence misses and will usually do much better. The trade-off is that dynamic strategies require some additional hardware support to handle marking bits. Static strategies require no hardware support other than the ability to invalidate cache lines under software control. Static strategies can be used on some existing machines, such as the BBN TC2000[21,23].

Figure 3.5 is an example of applying a (possible) dynamic strategy versus a static strategy to the code in figure 3.1. The dynamic strategy treats elements written this epoch, e.g. `A(1)` on p_1 after the first epoch, special and prevents them from being subject to the invalidate (represented by the line bypassing the invalidate before merging). `A(1)` is valid on p_1 during the second epoch. If the schedule of epoch 2 were reversed (i.e. p_1 gets iteration 2 and references `A(1)`), then reuse of `A(1)` would be realized.

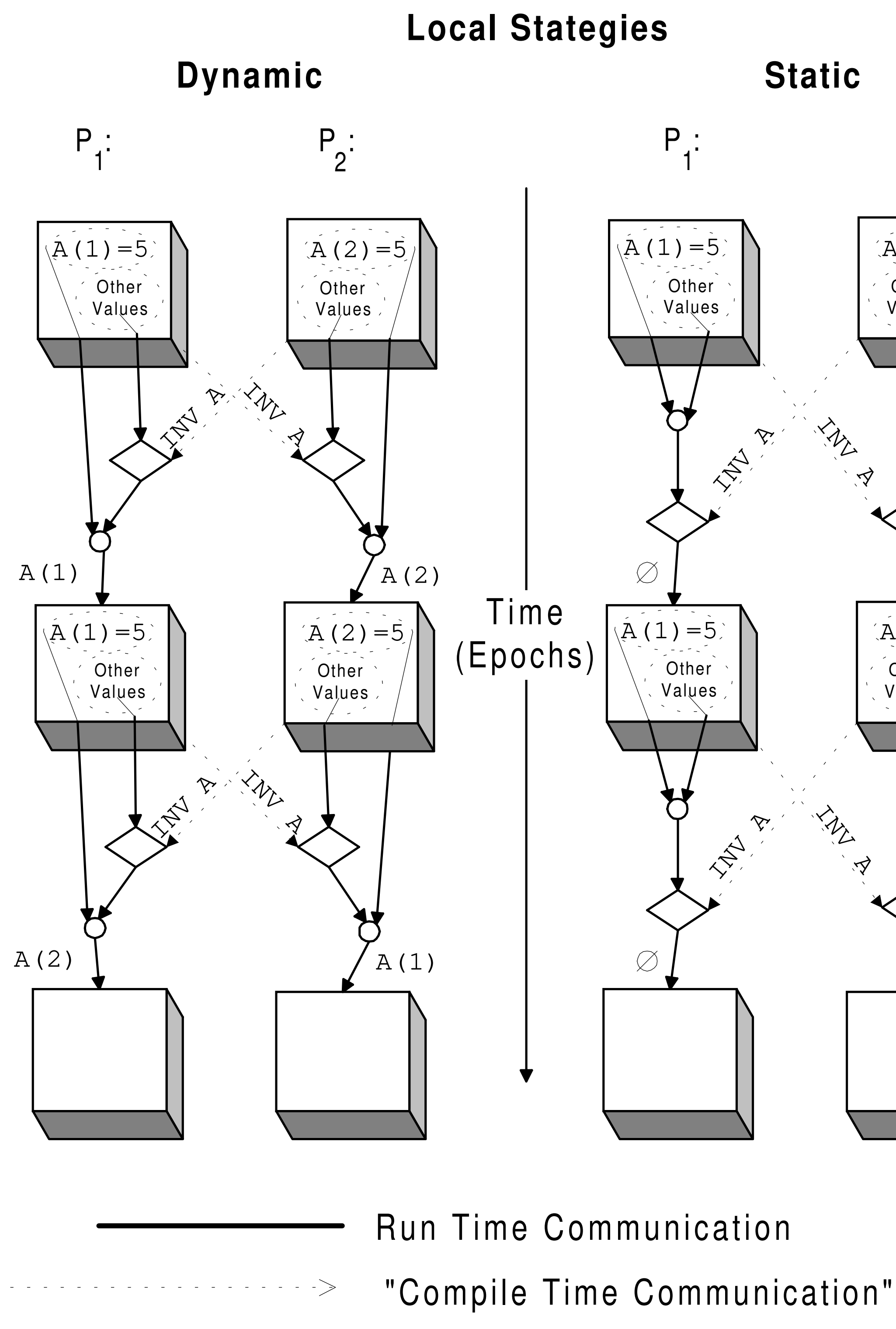


Figure 3.5: Dynamic versus Static Coherence

In the static strategy, the value of `A` written in the first epoch cannot be allowed to reach the third epoch. Since the second epoch *might* write `A(1)` on a different processor, an invalidate must remove the `A(1)` from cache that was loaded in the first epoch. That `A(1)` might be referenced on p_1 cannot be used to any advantage. The static strategy can only use what is known for sure at compile-time (represented by all of the cache contents being logically merged before the invalidate is applied). The figure is somewhat misleading because it is only necessary to invalidate `A` in one place in this restricted example. But whichever is chosen (after epoch 1 or after epoch 2), an unnecessary miss will be the result for some schedules. Thus, static schedules have an inherently higher miss ratio than dynamic strategies. Variables that are definitely not written in the epoch are not subject to invalidation (not shown in figure 3.5).

In this example, the invalidate was applied to a whole array and the exceptions made on an element by element basis. Both of these levels of granularity can be different while still preserving the essential flavor of an dynamic strategy.

3.2.4 Ideal Local Coherence

While local strategies cannot achieve the same ideal hit rate that global strategies can, there is an upper bound on the hit rate which any local strategy can achieve:

> Definition 3.7: Ideal Local Coherence
>
> A local coherence strategy such that all misses beyond those that would have occurred in an ideal global strategy:
> 1) Are compile-time stale (definition 3.3).
> 2) Cannot be proven to not be run-time stale by local information (definition 3.2).

Any particular strategy may fall well short of this goal. But, it does give a standard by which to measure different strategies. The definition is necessarily vague because the notions of what the compiler can prove and what can be known at run-time are also vague. In particular, compile-time analysis is relative to some level of granularity. If the compiler assumed that any write to an array writes to the whole array, that is a larger level of granularity (and correspondingly weaker 'ideal' strategy) than a compiler that tries to track individual elements. This definition is justified because 'ideal' refers to the coherence strategy itself and not to the quality of the compiler analysis. All ideal strategies will achieve the same hit rate for the same level of compiler analysis, regardless of their implementations. However, implementation considerations dictate that not all ideal strategies are equally appropriate for a given level of compiler analysis.

With some minor exceptions, definition 3.7 can be viewed as saying the following:

If (for some epoch e)
    $\mathtt{A}_w$ - i.e. appears to be written at compile-time and
    $\neg\mathtt{A}_w[\mathrm{i}]$ - i.e. is not accessed on p_i at run-time
Then
    $\mathtt{A}_w[\mathrm{j}]$ - must be assumed by p_i (i^1j)

Apparent writes (pessimistic compile-time information) which are not disproved by the local schedule theorem (3.5) (using exact run-time information), must be assumed to have happened. If they are part of an AT, then appropriate values must be invalidated. If not, the assumption of the write causes no harm. Obviously, this applies only to dynamic strategies. The example dynamic strategy used to explain figure 3.5 behaves this way, achieving ideal local coherence. Not all dynamic strategies need be ideal. In practice, achieving idealness requires either the run-time cost, the hardware implementation cost, or the granularity of the compiler analysis to go up (i.e. coarser analysis).

Even though the local schedule theorem (3.5) only discusses the immediately subsequent epoch, it is sufficient to achieve idealness. It is neither useful nor necessary to try to look beyond the current epoch into order to achieve idealness with respect to applying the local schedule theorem to override compile-time decisions. However, when the local schedule theorem says nothing, some (but not all) strategies do need to look beyond the current epoch to make a decision about whether or not an apparent write causes staleness.

The local schedule theorem is not the complete embodiment of what kinds of local knowledge might be useful (in part 2 of definition 3.7). Various degrees of replication of control flow are possible. p_i might have global knowledge of p_j's actions because p_i performs them too. In some circumstances, this can improve performance. In general, we ignore this possibility though some particular *ad hoc* exceptions are made where appropriate. Outside of this, the local schedule theorem captures reference patterns and control flow, i.e. all local information that can be used for coherence. For this reason, ideal local coherence can be rephrased in terms of the local schedule theorem (§7.5).

To understand this, consider the extreme case of a compiler with oracle quality dependence analysis. It knows exactly which values are written in a given epoch (though it still does not know the run-time schedule). If a given word, x, appears to be written in this epoch, but p_i does not observe it to have been referenced, then p_i knows that some p_j did write x. p_i invalidates x at the end of this epoch before it can possibly be referenced in another epoch. This is exactly what a global strategy would do, invalidate p_i's copy because p_j wrote it. If, on the other hand, p_i referenced x, it would be left in the cache in both cases.

It is worth noting that an ideal static local strategy is considerably more difficult to define (§6.3). What, if anything, it may be is not known to the author at this time.

Chapter 4

Previous Local Strategies

Despite the simplicity of the definition of coherence, implementations to realize it can be quite complex and diverse. In this chapter, we survey previous compiler strategies, noting their strengths and weaknesses, and placing them in our framework

4.1 Cytron, Karlovsky, McAuliffe

One of the first proposed compiler-assisted coherence strategies, the CKM strategy [21], used a direct attack on the problem by preceding any compile-time stale read with an invalidate for that value. It also addressed the converse problem, updates, i.e. when to move values from cache to main memory. The answer is simply that any write for which a read could occur on another processor needs to up updated after being written. As a analysis matter, in this limited framework, deciding when to update looks essentially the same as deciding when to invalidate. In general, deciding when to update is not as interesting a problem as invalidating and can be handled separately (section.).

CKM includes two optimizations. The first is that if all uses that follow a write are stale, the update that would otherwise follow the write can be a flush (update and invalidate) and the corresponding invalidates can be removed. For example, CKM would produce the following coherence augmented code:

```
X = A(i)
FLUSH (X)
PDO J
   IF (c1)
      =X
      A(J)=
      UPDATE (A(J))
   ENDIF
END
PDO J
   INVALIDATE (A(J))
   =A(J)                        This is not stale for c1 false
ENDDO
```

No invalidate is needed for the read of X in the loop. But, the update of A should not be made a flush because a possible non-stale reference follows in the last epoch.

One of the difficulties with this approach is that the invalidate instructions interfere with reuse on the same processor because of the uncertainty of control flow. This is handled with the optimization of moving invalidates to the start of the epoch (when address information is available). The most important aspect of this is hoisting invalidates out of serial loops where possible. An example of this optimization is:

```
PDO I
   INVALIDATE (A(I))
   INVALIDATE (B(f(I)))
   =A(I)+B(f(I))
   ...
   INVALIDATE (B(g(I)))
   =A(I)+B(g(I))
```

If the `INVALIDATE (A(I))` can be moved to the start of the epoch and handle both read references. However, for `B` this cannot be done because its subscript expression is not computable (in some cases) at the beginning of the loop. Therefore, some intra-epoch reuse must be sacrificed when *f*`(I)`=*g*`(I)`. This is the first weakness of CKM.

The paper also mentions the possibility of blocking invalidates but does not explore it. It considers the possibility of moving invalidates to alternate control flow branches to be used like prefetches and develops a data-flow framework for this analysis.

The second obvious weakness of this approach is the overhead involved in performing the invalidates. The most advantageous situation (overlooking unexplored blocking algorithms) is where the invalidate is a machine instruction. Even in this case, many references to stale values must have another machine instruction to perform the invalidate for each reference, thus doubling instruction execution costs for such references. On any currently existing system, invalidates are system calls with a much higher cost.

The advantage to this approach is that it is a static local strategy and can be made to work with no additional hardware (other than exposed coherence control instructions). It was early work in the area and should be viewed more as a demonstration of approach than an actually useful implementation.

4.2 Fast Selective Invalidation

Fast Selective Invalidation (FSI) [15] addresses the cost problem in CKM by using a small amount of additional hardware to perform the invalidates quickly. Each cache line

		Shared		Private
		Valid Bit	Change Bit	Valid Bit
State:	Fresh	1	1	n/a
	Valid	n/a	n/a	1
	Stale	0		0
Action:	Reference	1	1	1
	Invalidate (end of epoch)		0	n/a
Effect on Reference:	Hit	1	1	1
	Miss	1	0	n/a
	Miss	0		0

Table 4.1: FSI Summary

has a *change* bit as well as the conventional valid bit. The change bits are reset at every epoch boundary in O(1) time by a chip level reset (i.e. invalidated). Each program reference is marked by a bit in the instruction (or address part of the instruction), the ‘cache-read’ bit, indicating whether or not it is a (potentially) stale access. All accesses set the change bit. Non-stale accesses ignore the change bit. Compile-time stale accesses are hits only if the change bit is set, i.e. only if they have already been referenced in this epoch (since all change bits were last reset).

This is summarized in table 4.1. The 'state' section indicates which state the cache line is in based on the bits. The 'action' section indicates how the given action changes the bits. The 'effect' section indicates what the effect of a reference will be for the given bit combination. The valid state (as defined in 3.6) does not exist. However, the valid bit must still exist to handle non-shared references. Combing the valid bit and change bit would not be appropriate because it would no longer be possible to issue a chip level reset to effectively invalidate only shared data.

Figure 4.1 is an example of FSI. The first two epochs are not stale because the references to `A` cannot be the essential read of an AT (definition 3.1). Both references in third epoch could be stale and are marked as such in the instruction itself. Each of the first two references to `A` set the change bit in the cache line. However, before the third epoch is entered, all change bits are cleared. The first reference to `A` in the third epoch misses because its change bit is false (before the reference). This is appropriate because it could

Code	Marked as possibly Stale	A's *Change* bit in cache		Effect
		Before	After	
`PDO I`				
`=A(I)+B(I)`	no	0	1	
`ENDDO`				
`PDO I`				
`A(I)=`	no	?	1	
`B(I)=`				
`ENDDO`				
`CLEAR CHANGE`		1	0	
`PDO I`				
`IF (`*true*`)`				
`=A(I)`	yes	0	1	miss
`ENDIF`				
`=A(I)`	yes	1	1	hit
`ENDDO`				

Figure 4.1: FSI Example

be a stale access. The second reference to A in the third epoch is always a hit though because the change bit will be set, reflecting intra-epoch reuse.

The effect of this strategy is that all intra-epoch reuse is preserved (e.g. between the two reads of A in the third epoch) but all shared data is invalidated after every epoch. An important consequence of this is that the hit rate is the same no matter what actual schedule is used at run time. Thus, this has the same effect as a static local strategy, and is considered such in our framework. The extra hardware serves only to create an efficient invalidate.

Private and non-stale references are excluded and do not present a problem. The execution time overhead is negligible. There is some custom hardware required. The cost is quite modest, but it does prevent this from being applicable to machines without special hardware.

Even for a static local strategy, the problem is that this grabs too much. No inter-epoch reuse is preserved following epochs with essential writes (of ATs) to any value. Every stale value is invalidated instead of merely every AT. In the previous example, B is invalidated even though there is no need for that.

4.2.1 FSI – Non-unit Cache Lines

FSI is designed only for a write-through machine with unit cache lines. The logic of it depends on this in important ways. There is no marking bit to indicate whether or not a write can be stale. This implies that there is no read associated with a write and therefore only unit cache lines. The assumed machine is one where the actual write-through to main memory can occur sometime after the actual write into cache. Therefore, the write is never a miss nor does it incur a penalty. Writes do set the *change* bit in memory as do reads. The 'cache read' bit on the actual read access will determine whether or not the value left cached by the write should be considered a hit.

This does not have an obvious extension to a non-unit cache line machine. On such a machine, writes do incur miss penalties and do act like reads (for the purposes of coherence) because the whole cache line must be brought into cache when just a part of it is being written. This makes it difficult to compare FSI to other strategies, which assume that write misses incur penalties. For the purposes of such comparisons in this work, we assume that a reasonable model for extending FSI is that writes are marked with *change* bits as if they were reads and that write misses cost as would read misses.

This is not entirely fair to FSI. In some cases, the compiler can prove that none of the rest of the cache line which is loaded with a write will be referenced in this epoch. In that case, the 'cache read' bit can be set for the write such that it will hit even if that leaves stale data in the cache. For instance, in figure 4.1, the write of `A` in the second epoch can always be allowed to hit in cache, even though the extra words in the cache line are stale, because the eventual read in the third epoch will miss and get the new value from the second epoch. The default assumption is that the reads in the second epoch which are implicit in the write might be used in that epoch and thus the write needs to be forced to miss to get valid values. Instead of trying to extend FSI this far, we tested the unit cache line case under the zero-cost write-miss model for several methods in addition to the more standard non-unit cache line model.

4.3 Life Span Strategy

Life Span Strategy (LSS) [13] is built on FSI and improves on it. It directly implements the Local Schedule Theorem (theorem 3.5) with a special bit in each cache line that records the fresh state (called the *stale* bit in the original paper).

		Shared			Private
		Valid Bit	Change Bit	Stale Bit	Valid Bit
State:	Fresh	1	1	1	n/a
	Valid	1	1	0	1
	Stale	0	0	0	0
Action:	Reference	1	1	1	1
	Invalidate		Stale bit	0	n/a
Effect on Reference:	Hit	1	1		1
	Miss	1	0		n/a
	Miss	0			0

Table 4.2: LSS Summary

Like FSI, every epoch that writes to any global variable is followed by an invalidate that applies to all of cache. This converts all fresh cache lines to valid in the next epoch and to stale in subsequent epochs. The states and transitions are shown in table 4.2.

The FSI change bit mechanism is also used so that non-stale data and local values are not affected by the invalidate.

An example of LSS, with p_i getting iteration i on every epoch, is shown in figure 4.2. The invalidate-induced transitions are shown crossing epoch 4 (and omitted for clarity elsewhere).

In e_3, LSS catches the reuse FSI misses, the `A` that was written in e_2 can be left in the cache and used in e_3. The difficulty arises in e_5. `A` misses even though that is not necessary. The invalidate after e_3 made `A` merely valid (and no longer fresh) as it should have. However, the invalidate after e_4 (made necessary by the write to `B`) made `A` stale when that was not logically necessary. The granularity of compiler analysis is very coarse, any shared write is treated as if all shared variables were written.

The shortcoming was partially addressed in the paper by proposing extended LSS. In this method instructions include a field that codes how many epochs into the future a reference must surely remain valid. This field becomes part of the cache line and is decremented on every invalidate. It is implicitly 1 in simple LSS. In the previous exam-

Code	Reference marked as possibly Stale	A's *Change* bit in cache: Before	A's *Change* bit in cache: After	A's *Stale* bit in cache: Before	A's *Stale* bit in cache: After	Effect
PDO I						
=A(I)+B(I)	no	?	1	?	1	
ENDDO						
PDO I						
A(I)=	no	1	1	1	1	
B(I)=						
ENDDO						
INVALIDATE		1	1	1	0	
PDO I						
IF (*true*)						
=A(I)	yes	1	1	0	1	hit
ENDIF						
B(I)=A(I)	yes	1	1	1	1	hit
ENDDO						
INVALIDATE		1	1	1	0	
PDO I						
B(I)=					0	
ENDDO						
INVALIDATE		1	0	0	0	
PDO I						
=A(I)	yes	0	1	0	1	miss
ENDDO						

Figure 4.2: LSS Example

ple, the reference to A in e_3 could have its lifetime set to 2 epochs (at least) since no control flow path leaving e_3 rites A in the next epoch. This trades off hit rate for using more bits in the cache line. To efficiently perform the decrement, the paper suggests a unary representation and a shift register. Therefore, the number of extra bits needed equals the number of extra epochs to span (and not the log of that number). Also, it must be pessimistic about control flow and choose the minimum number of epochs. That is, control flow analysis must be based upon compile-time only knowledge.

If, in practice, values stay in cache for only a few epochs due to cache size limitations, a small number of bits can be used for the extended LSS at no great cost. Additional bits would not help because the values would already be evicted before the count runs out.

In summary, LSS retains complete intra-epoch epoch reuse like FSI. It also retains much inter-epoch reuse. It executes efficiently at run-time. However, the hardware cost, while still modest, is no longer trivial. In particular, there are no off the shelf components that perform the parallel shift required to implement the invalidate.

4.4 Parallel Explicit Invalidation

Parallel Explicit Invalidation (PEI) [47] seeks to gain the accuracy of only invalidating those arrays which are actually written while still using a fast invalidate. In this regard, it improves on LSS by never having an invalidate interfere with other arrays. The cache hardware must be able to invalidate every address which matches a particular bit mask (e.g. 0101xxxx means invalidate 16 bytes starting at address 50h). Arrays are laid out in power-of-2 configurations so that there is some bit mask which corresponds to each array and does not include part of another. This cannot be extended to work for array sections in general, though it can be made to work for some special ones. Finally, the invalidate is made part of every write. The particular word being written is excluded from invalidation (this is correct by theorem 3.5). The effect is that each write invalidates entries in this cache that have been made stale by the same instruction on other processors. Thus, it still acts on conservative compile-time information. When a write instruction does not execute on every processor, a dummy write must be added in order to perform the invalidate on every processor.

An example of PEI (with p_i getting iteration i) is:

Code	Mask	after reference: Processor 1 `A(1)`	after reference: Processor 1 `A(2)`	after reference: Processor 2 `A(1)`	after reference: Processor 2 `A(2)`	Effect
`INTEGER A(2)`	110	address of `A(1)`				
`PDO I=1,2`						
`   =A(I)`		valid	n/a	n/a	valid	
`ENDDO`						
`PDO I=1,2`						
`   A(I)=`	11x	valid	stale	stale	valid	
`ENDDO`						
`PDO I=1,2`						
`   =A(I)`						hit
`ENDDO`						

The only invalidate in this example happens implicitly with the write in e_2. When p_1 writes `A(1)` it simultaneously sends out the mask '11x' which invalidates all matching cache lines on p_1's cache except for `A(1)` (at address 110). In this contrived example, it leaves simply `A(2)` (at address 111). p_2 does the converse. Both processors then hit in e_3.

Where LSS needs an extra bit to preserve inter-epoch reuse, this achieves it with no extra storage by performing the invalidate immediately instead of waiting for the end of the epoch and then avoiding those lines just referenced. However, PEI would seem to sacrifice some intra-epoch reuse. Consider:

```
PDO I
   A(f(I)) =
   A(g(I)) =
   = A(f(I))
ENDDO
```

The programmer has guaranteed this is a legal program (e.g. because of the step of the `I` loop). However, PEI would sacrifice reuse on the read of `A(`*f*`(I)`). In those particular cases where a bit mask could differentiate the two references, this problem could be avoided (e.g. *f*`(I)` is always even and *g*`(I)` always odd). In the more common case where there is no `A(`*g*`(I)`), intra-epoch reuse is preserved.

In summary, PEI achieves array level resolution of invalidates, preserves inter-epoch reuse, and is fast. Its most serious shortcoming is the complexity of the invalidate. It requires enough extra instruction space for the mask and it requires a cache capable of associative matching. At the least, this implies a fully associative cache, an expensive proposition. Since the invalidate must occur in the same number of cycles (or nearly so) as a write, less aggressive implementations would not be satisfactory. The obligatory power-of-2 layouts for arrays can be wasteful of memory.

4.5 Time Stamping

Time stamping (TS) strategies [15,50] achieve ideal local coherence (definition 3.7) for a whole array granularity of analysis. FSI and LSS age cache lines with invalidates that move them from one state to another until they are finally stale. Instead of counting down in the cache line itself, TS counts up with a *clock* (a counter) duplicated on each processor for every array. The *clock* counts how many epochs have occurred in which a given array might have been written. That is, the *clock* for a given array is incremented at the end of any epoch which might have written the given array. The cache line itself contains a *time*

		Per Processor	
		Per Cache Line	Per Array
		Time Stamp	Clock
State:	Fresh	clock+1	
	Valid	clock	
	Stale	<clock	
Action:	Write	clock+1	
	Read w/ Write in Epoch	clock+1	
	Read w/o Write in Epoch	clock	
	Epoch w/ Write		+1
Effect:	Hit	^{3}clock	
	Miss	<clock	

Table 4.3: TS Summary

stamp which is set to the current *clock* + 1 when the line is referenced in an epoch with a write. If there is no write in an epoch, the *time stamp* is set to *clock* (because the *clock* will not be incremented at the end of the epoch). This implements theorem 3.5.

The time stamps stay unchanged through the life of the cache line. When the line is referenced again, its time stamp is compared to the current clock to determine if the line is stale. The effect of this is to implement exactly the three states shown in figure 3.4. The effect is summarized in table 4.3. The clocks are duplicated for each processor. All processors increment the same clocks at the same time (as measured by epochs) based on compile-time analysis. Private data is handled in the usual way and not shown in the table since it is completely separate.

The utility of this is that it is not necessary to simultaneously perform some action for every cache line. The state implicitly changes by reference to the appropriate clock. The actual detection of staleness is delayed until the line must be referenced anyway. When a variable is referenced and its time stamp set to *clock*+1, it will obviously be a hit if referenced again in this epoch.

Figure 4.3 is an example of TS. The schedule is `A(I)`[i] and `B(I)`[i]. `A(2)` is not relevant to what happens on p_1 and is omitted for space. In e_1 (epoch 1), there are no writes. `A` & `B` are both loaded with their cache time stamps set equal to their current clock value, 0. In e_2, the write of `A(1)` sets the time stamp to 1 (clock+1), in anticipation of the increment of `A`'s clock a the end of the epoch. During the rest of e_1, `A(1)`'s time stamp > `A`'s clock so `A(1)` is fresh. `B` is similar. In e_3, the reference to `A(1)` finds it in cache, extracts its time stamp and compares it to the clock value. Finding them both equal to 1, `A(1)` is valid (no longer fresh) and a hit. This applies regardless of the control flow

	Processor 1 (post reference states)						
	A		B				
	A(1) Time Stamp	A Clock	B(1) Time Stamp	B(2) Time Stamp	B Clock	Effect	Epoch
		0			0		
`PDO I=1,2`							1
`   =A(I)+B(I)`	0		0	--			
`ENDDO`							
`PDO I=1,2`							
`   A(I)=`	1						2
`   B(I)=`			1	--			
`ENDDO`							
`INVALIDATE A,B`		1			1		
`PDO I=1,2`							
`   IF (`*true*`)`							
`      =A(I)`	1	1				hit	3
`   ENDIF`							
`   B(I)=A(I)`	1	1	2	--		hit	
`ENDDO`							
`INVALIDATE B`					2		
`PDO I=1,2`							
`   B(3-I)=`			2	3			4
`ENDDO`							
`INVALIDATE B`					3		
`PDO I=1,2`							5
`   =A(I)`	1	1				hit	
`   =B(I)`			2	3	3	miss	
`ENDDO`							

Figure 4.3: Time Stamp Example

or additional references in the same epoch. The write to `B(1)` sets its time stamp to 2 (clock+1) so it will be valid next epoch. `A` is not affected because only `B`'s clock will be incremented at the end of the epoch (in contrast to LSS which would affect both).

In e_4, the reference pattern is reversed. On p_1, `B(2)` is loaded into cache with a time stamp of 3, indicating it will be valid next epoch. `B(1)` stays in cache with its previous time stamp of 2. In e_5, the reference to `A(1)` is a hit because it was referenced on the same processor in e_3. The intervening write to `B` in e_4 has no effect because the clocks for `A` and `B` are separate. In contrast, the read of `B(1)` finds it in cache with its time stamp set to 2, which was the last epoch on which this copy of B was written (counting only epochs that possibly write to B). The clock for B is 3, indicating that B might have been written since this copy was loaded in cache. Therefore it misses.

TS strictly improves on CKM, FSI, & LSS for hit ratios and is an ideal dynamic local strategy (theorem 3.7), at the whole array level of granularity. At finer levels of granularity, it still misses possible reuse. For instance,

```
PDO I=1,N
   A(I)=
ENDDO
A(N)=
PDO
   =A(I)
ENDDO
```

would need an invalidate for all of A after the second epoch even though only one element was written. This would cause all of A(1:N−1) to miss in the third epoch even though that is not necessary. PEI will capture some of this inter-epoch reuse that exists at finer levels of granularity. But, it also sacrifices some intra-epoch reuse and is difficult to categorize as clearly better or worse than TS.

4.5.1 TS Control Flow

TS has another important advantage, besides granularity, over extended LSS. The array clocks are not incremented until an epoch is actually executed. Thus, the compile-time analysis does not need to know anything about inter-epoch control flow. The implications of that control flow are entirely delayed until run time. LSS must statically decide how many epochs a reference will be valid for which requires it to determine, as best possible, control flow at compile time.

4.5.2 TS Hardware

TS has the following hardware requirements:

- A time stamp in every cache line.
- A clock designator in every shared memory reference.
- Fast access clocks.
- Fast '<' & '>' comparisons (versus only '=' for tag matches)

When a cache access occurs, it is necessary to decide which clock to compare the cache line time stamp against. This is done by using a designator in the reference to find the right clock. This can be done simultaneously with fetching of the cache contents and tag. The tag comparison can be done at the same time as the clock & time stamp comparison. This may need to be duplicated for each cache set (i.e. the set associativity), typically 2 or 4. Extending TS to handle a granularity finer than whole array is problematic because there would need to be a clock for each section being considered and additional sufficient bits in the instruction to code for all of the possible different clocks. This is covered in more detail in section 8.2.2.

This is still considerably cheaper than PEI because the comparator exists only once, or perhaps once for each possible block (set associativity), 2 or 4 being typical. PEI needs a '=' comparator for every cache line.

There is a peculiar limitation to TS. The clocks can overflow. When that happens all cache lines which depend on that clock must be invalidated. This is the same problem LSS suffers. Time stamping uses binary counters and the impact is much less. Also, the overflow in TS is based on the actual control flow path taken (above the epoch level) instead of worst case compile-time control flow as in LSS.

4.6 Summary

Previous local strategies all suffer from the inability to exploit the fullest resolution of compile-time analysis. We summarize all strategies by their important characteristics after presenting our new ones (§9.6).

Chapter 5

CTV – Coherence Through Vectorization, A Static Strategy

5.1 Introduction

Some machines provide caches only for local data and make no attempt at coherence. The intention may be that shared data simply has to be kept in global memory or that it be managed explicitly. In either case, neither global coherence nor hardware assisted coherence is available.

CTV is a technique applicable to such situations, i.e. a static local strategy. It does require that INVs (invalidates) and UPDs (updates) be exposed to the compiler as available cache control operations. These can often be simulated even when there is no coherence hardware *per se* (by accessing dummy values that map to known cache slots). Regardless, all is lost if this bare minimum of support is not available.

CTV starts by annotating every reference which needs invalidation with an INV instruction. This is essentially the CKM approach [21]. It is obviously correct, but not efficient. CTV proceeds by 'vectorizing' the INV's. That is, it tries to hoist the INV's as far out of loops as possible. The first way it can do is by recognizing simple loop invariance. This directly saves invalidate overhead and improves the hit ratio by avoiding redundant invalidates. The second way it can hoist INV's is by recognizing that a set of INV's all refer to contiguous memory locations. They can then be grouped into a single system call that operates on the whole range of addresses more efficiently than a series of calls for each word would.

In the context of the CTV approach, deciding when to update can be managed similarly to deciding when to invalidate. For machines which efficiently handle writes, CTV need not do anything. Either write-through or write-back before synchronization are appropriate in the CTV model. But for those machines with expensive writes, the CTV algorithm specifies where to perform the actual write-back. This improves performance for essentially the same reasons as hoisting INV's.

Section 5.2 provides an overview of data dependence as needed by CTV, discusses some of the trade-offs to consider in such a static strategy, and provides a metric appropriate to such circumstances. Section 5.3 discusses how FSI and CKM compare based on these particular criteria relevant to static strategies. Section 5.4 presents some basic background on vectorization and then the details of how the CTV coherence strategy is built on top of that. Using an example, we explain the behavior of our algorithm and differentiate its results from those of other compiler-based coherence strategies described in the literature. Section 5.5 presents some experimental results on the BBN TC2000 that compare the performance of three application kernels using out CTV approach versus the CKM and FSI approaches. CTV is shown to be superior to CKM. FSI requires special hardware and depends on the cache being write-through. Despite this, CTV can still perform better or comparably with FSI on the TC2000.

5.2 Overview

Static coherence strategies often (though not necessarily) see coherence as a matter of *processor crossing dependences* [21]. CTV takes this view. Processor crossing dependences are less general than Access Triples (ATs), but it is much easier to generate them during dependence analysis (no section analysis is needed).

A data dependence exists between two statements, S_1 and S_2 if there is a path from S_1 to S_2 and both statements access the same location in memory. CTV is concerned with three types:

- True dependence — S_1 is a write and S_2 is a read.
- Anti dependence — S_1 is a read and S_2 is a write.
- Output dependence — S_1 is a write and S_2 is a write.

A data dependence between statements S_1 and S_2 is *carried* by a loop if the execution of S_1 in loop iteration i can potentially access the same memory location as the execution of S_2 in loop iteration j, $i \neq j$. The nesting depth of the outermost loop which carries a dependence is said to be the *carrying level* of the dependence. By convention, dependences which result from sequential code execution (are loop independent) are said to be carried at level ∞.

A processor crossing dependence is simply a one for which each end of the dependence can be in different epochs (executed on different processors). This is true exactly when its carrying level is less than the parallelism level of either of its ends.

Compiler-based algorithms for software cache coherence are based on the following principle: processor-crossing true dependences must be augmented with coherence operations to ensure that values flow between processors. The WRITE in a processor-crossing true dependence must be followed by an UPD to send the new value to main memory; the READ must be preceded by an INV if it is possible that the cache contains a stale value. To ensure values flow properly when a true dependence is present, previous work has focused on placing an INV between the write of a value and its subsequent read.

Like previous approaches, we consider programs with loop-based parallelism that do not contain explicit synchronization. In CTV, as in other static coherence strategies, we focus on maintaining coherence based on processor crossing dependences and reserve a more general treatment of access triples for later chapters. For the rest of this chapter, we only consider placement of INVs between the write and subsequent read in a true dependence. We assume that the run-time mapping of parallel loop iterations to processors is unknown at compile time. Furthermore, we assume that parallel loops do not carry any true-, anti-, or output-dependences. Such dependences indicate that the data written by one loop iteration is not independent of the data accessed by other iterations of the loop (i.e., race conditions exist). For programs that satisfy this restriction, a processor crossing dependence must flow across the start or end of a parallel loop. In particular, a dependence crosses processors when it is carried by a serial loop outside of a parallel loop, or when it links a pair of statements with one of the statements nested inside a parallel loop and the other statement in a serial region or a different parallel loop.

For expository purposes, consider each serial section (a code section not enclosed by any parallel loop) to start with a dummy FORK and end with a dummy JOIN. With this assumption, the sequence of operations to ensure proper coherence for a processor-crossing true dependence is WRITE, UPD, JOIN, FORK, INV, READ. Operations on unrelated data can be interleaved arbitrarily within this sequence. If complete knowledge were available, the best spot to place the INV would be immediately after the FORK. Moving the INV closer to the READ may remove an opportunity for reuse of a value since there may be some other reference that brings the value into cache before the INV.

Moving the INV closer to the FORK to facilitate reuse can make it difficult to issue a precise INV, particularly if the INV is moved to a point prior to calculation of subscripts used to determine the location of the READ. Such imprecision does not affect correctness as long as the region covered by the INV subsumes the data accessed by the READ. This leads to a tradeoff affecting potential reuse of cached values.

Figure 5.1 illustrates the sort of tradeoffs that occur in the placement of INVs. The `PARALLEL` construct corresponds to a fork-join pair that specifies a block of code to be executed on every processor. `PDO` is a work-sharing construct; iterations of a `PDO` are partitioned among the processors for execution. If the INV for reference 2 is placed immediately after the FORK (option 1), it must invalidate all of `A` since at that point it cannot be determined what part of `A` reference 2 will access. This invalidate will eliminate most reuse of `A` by reference 3 in the trailing serial loop. Alternatively, if the INV for reference 2 immediately precedes the reference (option 2), `A(I)` is invalidated ten times, once for each iteration of the `J` loop, which is clearly unnecessary. However, this placement of the INV does not disturb reuse of `A` by reference 3 in the trailing serial loop since most of the INVs will take place on a processor different from the one that executes the serial code.

```
REAL A(10)
DO I=1,10
A(I)=  ...
ENDDO
PARALLEL
INV  A(1:10)     [option 1]
PDO I=1,10
DO J=1,10
   IF (MOD(J,2).EQ.1) THEN
      INV  A(I)  [option 2]
      B(I) = A(I)   [reference 2]
   ENDIF
ENDDO
END PDO
END PARALLEL
DO I=1,10
   ...=A(I)    [reference 3]
ENDDO
```

Figure 5.1: Placement of Invalidate

When evaluating the effectiveness of a caching strategy, people most often focus on the hit ratio. The hit ratio is only half of the picture though. The hit ratio measures how effective reads are, but does not measure how effective writes are. The read miss ratio (one minus the hit ratio) is how often a read misses in cache causing lost performance. A complementary measure is the write miss ratio, the percentage of writes which are made to main memory (instead of just cache) relative to the total number of writes. Both the read and write miss ratios are necessarily greater than zero: there is some intrinsic number of reads from main memory and writes to main memory that a program requires. We are interested in how well coherence schemes perform relative to this standard. We define two efficiency measures, the Cache Read Efficiency, CRE, and the Cache

Write Efficiency, CWE (table 5.1). These measures reflect the effectiveness of cache organization (e.g. associativity), but more importantly, how well a given coherence scheme does relative to an optimal scheme (for a given data set and number of processors). Optimal means the minimum number of main memory reads that a program execution would require given no evictions due to cache organization or size. Some of these reads are needed to bring values into cache initially. Others are needed because a value was changed on another processor and needs to cross processors.

R_t = total reads in program
R_o = optimal number of main memory reads
R_a = actual number of main memory reads

W_t, W_o, and W_a are defined analogously for writes

$$CRE = 1 - \frac{R_a - R_o}{R_t - R_o}$$

$$CWE = 1 - \frac{W_a - W_o}{W_t - W_o}$$

Table 5.1: Definitions of coherence algorithm efficiency.

The CRE is not the same as the hit ratio. The CRE is how the performance of a given coherence algorithm and cache compares to the ideal for a particular program execution. Minor changes in the program under study (e.g., register allocation of some array references) might make large changes in the actual `hit ratio' without changing the CRE. For some algorithms, R_t=R_o, in which case CRE is undefined. This is not just a mathematical triviality, but a reasonable interpretation. In this circumstance, no improvement is possible regardless of the coherence algorithm. Similar observations apply for the CWE.

5.3 Related Work

Here we re-examine the parts of CKM and FSI that are relevant to the particular issues raised in this chapter.

Both methods issue UPDs immediately after their corresponding WRITE. The FSI method does this implicitly by using write-through caching. The CKM method uses copy-back caching and thus requires explicit UPDs. However, the two methods treat INVs in completely different ways. The FSI method moves all INVs for a parallel region to immediately follow the FORK at region entry. FSI does not try to determine which locations need to be invalidated; it invalidates everything that is a shared writable object after every FORK, even if it is not referenced before the next JOIN. The CKM method goes to the opposite extreme and places every INV immediately before its corresponding READ. To decide which READs need invalidates, the CKM method uses dependence analysis to determine which READs are involved in processor crossing dependences.

Both papers make part of the access triple observation and use it in their methods to the extent of not placing INVs before READ references which follow only WRITE references. This is useful for FSI only if it applies to every reference in a loop. CKM can apply it on a reference by reference basis.

Cheong and Viedenbaum's simulations of the FSI method show that it has a good CRE, approaching 100%, but its CWE is always 0%. There is no data on how the CKM method fairs but one would expect it to usually have a lower CRE and a CWE slightly above 0%.

Cytron, Karlovsky, and McAuliffe discuss replacing UPDs and INVs with FLUSHes under certain circumstances. The treatment was apparently only for scalars and is not as general as it might be. They also suggest that coherence overhead could sometimes be reduced by moving INVs to a pre-dominator and UPDs to a post-dominator. A precise algorithm is not given. INVs can be moved to particular control flow branches so long as the other branches have an assignment to the variable. Without array kill information it is unlikely that these optimizations can be applied for subscripted variables. Scalars are not of interest since they are either local or read only; otherwise, the parallel loop would carry dependences thus violating one of the fundamental assumptions.

5.4 Vectorization Algorithm

Conceptually, our approach starts with the solution generated using the CKM method and applies vectorization [4] to aggregate cache control operations and move them as far as possible, toward either the FORK or the JOIN. The resulting INVs often resemble those created by the FSI method. We refer to our strategy as *Coherence Through Vectorization* (CTV). It maintains exactness in the sense that it never invalidates anything that does not need to be invalidated. And, of course, it never updates anything that does not need to be updated. It never pays the cost that the FSI method does of invalidating the wrong value. In the worst case CTV will pay the cost that the CKM method does of invalidating the same value too often, but it many cases redundant INVs will be avoided.

CTV has the additional benefit of reducing run-time overhead by aggregating cache control operations and thus amortizing their initiation overhead. For the UPD case, this

```
DO I=1,N
   DO J=1,N
      DO K=1,N
         DO L=1,N
            INV A(I,F(J),K)
```

Figure 5.2: Level of aggregation

improves on both the CKM and FSI method.

For the INV case, CTV does better than the CKM method, which does not aggregate READs. But, CTV will usually not aggregate as much as the FSI method, which lumps every INV for a given loop into a single global INV.

5.4.1 Vectorization Background

We use vectorization here to mean a particular process of reconstructing a program from its dependence graph. Generating vector code is not the objective *per se*. Vectorization algorithms determine how many levels of aggregation a statement has and restructure computations so that the aggregation can be realized. Such algorithms work entirely from a program's dependence graph and deliberately ignore the program's original structure.

We use the term *aggregation* in a special sense. The level of aggregation is the number of serial loops out of which a statement can be hoisted. This might be possible due either to loop invariance or the presence of a discernible section. If we were actually generating vector code, this would be the number of dimensions of vector parallelism. In figure 5.2, the INV has two levels of *aggregation*, `K` and `L`. The `L` level exists because the INV is invariant with respect to it. The `K` level exists because the section can be analyzed. No section can be constructed for the `J` level because of the unanalyzable subscript. Unless the `I` and `J` loops can be interchanged, there is no aggregation at the `I` level because the `J` loop is nested inside of it. The two levels of aggregation could be realized by changing the `INV` to `INV A(I,F(J),1:N)` and hoisting it out of the `K` and `L` loops.

5.4.2 The Algorithm

The essence of the CTV algorithm is to add INVs to the dependence graph in such a way that the INV is constrained to occur after the FORK, before the READ that needs invalidation, and as soon as any subscripts needed to determine the memory location are known. Other dependences on the variable needing invalidation are *not* relevant to the placement of the INV. The placement of the INV is not specifically determined; it is implicit in the structure of the dependence graph. The vectorization algorithm then restructures the code so as to achieve as much aggregation as possible for the INV. A similar process applies for WRITEs and UPDs.

1. Analyze the program for data dependence.

2. Determine which of the dependences are processor crossing.

3. For all references which have processor crossing dependences,
   (a) Create a coherence statement with the same reference.
   (b) Add dependences to insure than INV's precede their corresponding reference and that they follow the FORK for the loop the reference is in. UPD's are similar.

4. `CodeGen`, i.e. `Vectorize'
   (a) Collapse `PDO`s and dependence cycles to single nodes, thus forming a DAG.
   (b) Generate code on this DAG in topological order.
   (c) For each collapsed node, recurse by calling `CodeGen` at the next deeper nesting level.
   (d) For single statements, produce the text of the statement with the proper amount of aggregation.

Figure 5.3: CTV algorithm

The algorithm is summarized in figure 5.3. The rest of this section discusses it. Consider all `PDO`s to be collapsed into `PARALLEL DO`s for purposes of discussion.

In a preprocessor step, all array references are rewritten using temporary variables in order to make the subscript references side effect free. Next, we apply conventional dependence analysis to construct a dependence graph for the program. The dependence graph serves as the program representation for the remaining steps of the algorithm.

The next step is to determine which of the dependences are processor crossing. This can be easily determined by comparing the depth of any enclosing `PARALLEL DO`s, the common nesting level of both ends of the dependence, and the carrying level of the dependence.

Next, INVs are added. For each READ reference in the program, e.g. `A(t1,t2...)`, that is the sink of a processor crossing dependence (i.e. only true dependence is relevant), a statement to invalidate the same reference is added, e.g. `INV A(t1,t2...)`. A dependence from the INV to the READ is added to the dependence graph. If the READ is in the scope of a `PARALLEL DO`, a dependence is added to place the INV in the same `PARALLEL DO` as the READ. The INV is constrained only to stay in the same parallel loop as the

READ, not the serial loop(s) nested inside of the parallel loop. If the READ is in a serial section (not in the scope of a parallel loop), dependences are added to execute the INV after any `PARALLEL DO`s which are the sources of processor crossing dependences reaching the READ (this can be trivially extended for nested `PARALLEL DO`s). This essentially means that invalidation can occur any time between processor assignment in the FORK and the actual reference, regardless of other program structure and dependences. Finally, any dependences that existed on the subscripts of the READ, `t1, t2` ..., are copied to references to those subscripts in the new INV statement. This assures that any computations which are needed to determine subscript values have already been done. Other dependences on `A` itself are *not* copied. At this point, the placement of the INV for the reference and the level of aggregation available is determined solely by the single reference.

UPDs for WRITE references which are the source of processor crossing true dependences are added next. This is done analogously to the adding of INVs for READs. For anti- and output- dependences from WRITEs, an INV is added instead (unless there is already an UPD). This is necessary because a dirty value that is no longer needed after a task might be evicted by the cache hardware at some later time when space is needed. This value could then overwrite a more recent value in main memory.

In the circumstance where a READ follows a WRITE in the same task and both access *some* of the same locations, the strategy as outlined above might produce the following sequence of events: WRITE, INV, UPD, READ. When the INV and UPD reference overlapping sections, a dirty value would be invalidated before it is updated. An *inversion dependence* is added in this step to prevent this problem. Adding coherence statements for all of the READs before any of the WRITEs makes it easy to determine when inversion dependences are necessary. This can be done is such a way as to not affect the overall time complexity of the algorithm or hinder any possible aggregation that would otherwise be available.

We refer to a dependence graph augmented with coherence operations as a coherence graph. A coherence graph contains all of the dependences which must be satisfied for a correct execution of the program on a shared memory multiprocessor without hardware coherence.

Code generation, `CodeGen`, proceeds by collapsing all cycles in the coherence graph and all `PARALLEL DO`s (but not serial DOs) to single nodes. The resulting DAG is then processed in topological order. Different node types are handled in different ways. PARALLEL DOs must be collapsed to single nodes to keep them from being distributed. This insures that INVs will be done in the same task as their corresponding READs.

If a given node contains more than one statement, an appropriate type of DO statement is generated at this level and the contents of the cycle are processed by recursively calling CodeGen at one level deeper. In the recursive call, all statements and dependences outside of this node are ignored. Also, any dependences carried at the current level are ignored because they are satisfied by the DO. If the node is a single statement, code can be directly generated for it. The four cases are:

- Cycle: this indicates that the statements in the node originally came from a serial loop and they all depend on each other in some way including at least one loop carried dependence. This does not cause all of the statements originally in the DO to be generated, only those in the cycle.

- PARALLEL DO node: Generate a PARALLEL DO and then recurse.

- A single non-coherence statement: Generate it. If the current level is less than the original level of the statement then additional serial DOs will be necessary. This may occur because a statement is nested in a loop but has only loop independent dependence upon it. There are no cycles in this case, but the surrounding DOs are still necessary.

- A single coherence statement: Generate the coherence statement with a level of aggregation which is the difference between the current level of code generation and original nesting depth of the coherence statement. This is sufficient to ensure that coherence statements are generated at the outermost level possible, i.e. with as much aggregation as possible.

As a practical matter, cycles and single non-coherence statements on the same level will be fused together into one serial loop when doing so does not capture a coherence statement between them. This simply prevents loops from being distributed unnecessarily.

Non-unit cache line sizes cause aliasing of values. This problem must be addressed by any coherence scheme. If a parallel loop index does not appear in the fastest varying subscript of an array reference, it is not a problem. When it does, other compiler techniques such as changing the array layout or strip mining can be used to solve it. Failing that,

```
1     PARALLEL DO I=1,N
2           J2=0
3          DO J=1,N
4               J2=J2+1
5               DO K=1,N
9                    C(I,J) = C(I,J2) + A(I,K)*B(K,J)
```

Figure 5.4: Matrix Multiply before coherence

```
1     PARALLEL DO I=1,N
2          J2=0
3          DO J=1,N
4               J2=J2+1
5               DO K=1,N
6                    INV A(I,K)
7                    INV B(K,J)
8                    INV C(I,J2)
9                    C(I,J) = C(I,J2) + A(I,K)*B(K,J)
10                   UPD C(I,J)
11              END DO
12         END DO
13    END PARALLEL DO
```

Figure 5.5: Matrix Multiply after CKM coherence

either no-caching or write-through caching must be used for that variable inside any parallel construct in which this situation occurs. If the architecture does not permit such an allocation, some of the parallelism will have to sacrificed for a completely automatic technique. This must be handled before the coherence graph is built. A more detailed treatment of the algorithm can be found in [23].

5.4.3 Example

Consider a simple matrix multiply (figure 5.4) where the outer loop is parallel. The `J2` assignment in statement 4 is added for the sake of example (assume that the compiler does not recognize auxiliary induction variables). References to elements in arrays `A`, `B`, and `C` must all be invalidated before they are used because their initial assignments may have occurred on a different processor. The new value of `C` must be updated from cache to main memory before the `PARALLEL DO` loop finishes.

The FSI approach would solve this problem by invalidating the whole cache for each processor when the `PARALLEL DO` starts and using write-through so that every assignment to `C` in statement 9 goes straight to main memory. The problem with this is that for any given `I` and `J` it writes `C(I,J)` N times (for each iteration of the `K` loop instead of just once).

The naïve CKM approach would solve the problem by specifically invalidating before every READ and updating after every WRITE (figure 5.5). This suffers from the same problem as the FSI method for both the WRITE and the READ of `C(I,J)`. In this particular case, the INV and the UPD for `C` contain a loop invariant expression and could be hoisted. The original CKM paper discusses this possibility. But, that is still not good enough.

CTV conceptually starts with what the CKM approach produces (figure 5.5). Dependences are added from the `PARALLEL DO I` to the `INV A` and from the `INV A` to the reference to `A` in the assignment statement. Dependences for references to `B` and `C` are added similarly. Note that the dependences do not pin the coherence statements inside the inner `K` loop.

Next, `CodeGen` is invoked on the resulting coherence graph. The only node is the `PARALLEL DO` itself (other statements are collapsed into this node). That statement is generated. Then `CodeGen` recurses on the contents of the loop. The `Inv A` has no predecessors in the coherence graph so it can be generated, even though neither the `J` loop nor the `K` loop have been generated yet. The `Inv A` was originally at nesting level 3 and code is now being generated at level 1, so there are two levels of aggregation available. These are found and the final `Inv A` is generated. The `B` case is similar.

The `Inv C` cannot be generated yet because it depends on the assignment of `J2`. The `J2=0` assignment has no dependence predecessors and can be generated. All of the other statements which reference `J2` are in a cycle because of loop carried anti dependence. That cycle is handled by generating the `DO J` loop then recursing on its contents. The `INV C` can then be generated after the `J2=J2+1` which it depends on. Thus, the `INV C` is hoisted out of the `K` loop, but not the `J`. The final results is shown in figure 5.6.

```
1     PARALLEL DO I=1,N
2          J2=0
6          INV A(I,1:N)
7          INV B(1:N,1:N)
3          DO J=1,N
4               J2=J2+1
8               INV C(I,J2)
5               DO K=1,N
9                    C(I,J) = C(I,J2) + A(I,K)*B(K,J)
11              END DO
12         END DO
10         UPD C(I,1:N)
13    END PARALLEL DO
```

Figure 5.6: Matrix Multiply after CTV

A rough comparison of the three methods can be made by counting the READs and WRITEs to main memory versus cache (table 5.2). CKM+ refers to the results of the CKM method after making the simple optimization of hoisting loop invariant coherence instructions. Looking at CTV, every write on line 9 (figure 5.6) goes to cache since write-back is being used. There are n^3 writes evenly distributed across all p processors, so the time per processor is $(n^3/p)C_w$. For reads, each variable needs to be considered separately. `C(I,J2)` is invariant in the `K` loop but will be invalidated for each iteration of the `I` and `J` loop. Therefore, it will miss and be a memory read n^2 times. The remaining n^3-n^2 references will all be cache hits. This work is evenly distributed amongst the processors. `A(I,K)` is similar to `C(I,J2)`. `B(K,J)` is invariant for the `I` loop but it must be invalidated inside of that loop. Therefore, it will read all n^2 values from main memory for *each* processor. The total number of main memory reads is $n^2 + n^2 + p*n^2$, or $2n^2/p + n^2$ for each processor. The number of cache reads is simply that total number of references $3n^3$ minus the number of memory reads. For matrix multiply, this is the same as for the optimal case so the CRE for CTV is 100%. The other entries follow similarly.

Method	Reads	Writes
FSI:	$\left(\frac{2n^2}{p}+n^2\right)M_r+\left(\frac{3n^3}{p}-\left(\frac{2n^2}{p}+n^2\right)\right)C_r$	$\frac{n^3}{p}M_w$
CKM:	$\frac{3n^3}{p}M_r$	$\frac{n^3}{p}C_w$
CKM+:	$\frac{2n^3+n^2}{p}M_r+\frac{n^3-n^2}{p}C_r$	$\frac{n^3}{p}C_w$
CTV:	$\left(\frac{2n^2}{p}+n^2\right)M_r+\left(\frac{3n^3}{p}-\left(\frac{2n^2}{p}+n^2\right)\right)C_r$	$\frac{n^3}{p}C_w$
Optimal:	$\left(\frac{2n^2}{p}+n^2\right)M_r+\left(\frac{3n^3}{p}-\left(\frac{2n^2}{p}+n^2\right)\right)C_r$	$\frac{n^2}{p}M_w+\frac{n^3-n^2}{p}C_w$

Method	Invalidates	Updates	CRE (%)	CWE (%)
FSI:	N/A	N/A	100	0
CKM:	$\frac{3n^3}{p}(S+I)$	$\frac{n^3}{p}(S+U)$	0	0
CKM+:	$\frac{2n^3+n^2}{p}(S+I)$	$\frac{n^2}{p}(S+U)$	$\frac{n-1}{3n-p-2}$	100
CTV:	$\frac{n}{p}(S+nI)+\frac{n}{p}(S+n^2I)+\frac{n^2}{p}(S+I)$	$\frac{n}{p}(S+nU)$	100	100
Optimal:	N/A	N/A	100	100

- C_r Cache Read time
- C_w Cache Write time
- M_r Memory Read (includes cache fill) time
- M_w Memory Write (includes cache fill) time
- S Start a cache coherence operation (time)
- I Invalidate a cache entry (time)
- U Update a cache entry (time)
- n Size of Matrix
- p Number of processors, assume n mod p ≡ 0

Table 5.2: Cache Utilization of Different Methods on Matrix Multiply

The CKM and CKM+ method are clearly slower than the CTV method for this case. For machines without special write buffering , it is usually the case that $I << C_r \leq C_w < M_r \leq M_w \approx U$. S can vary widely depending on implementation. For this case, the CTV method is faster than the FSI method when:

$$\left(M_w - C_w\right)n^2 > \left(S + M_w\right)n + 3S$$

There is some n sufficiently large that the CTV method is better than the FSI method. the greater the difference between main memory and cache, the better for CTV. The greater the cost of the software overhead, the worse for CTV. In this case, the CTV method will do better than FSI by virtue of improving the CWE.

Efficient handling of writes (e.g. via write-back buffers) can reduce the apparent M_w to C_w. In those cases, FSI will perform better than CTV on matrix multiply by moving the software overhead cost of invalidates into hardware implementation cost.

5.5 Experimental Results

The CTV method, the CKM method, and the FSI method were applied manually to three small, simple programs which were tested on a BBN TC2000, a distributed shared memory multiprocessor, to evaluate their effectiveness. The three programs were a matrix multiply, a blocked LU decomposition, and a heat flow relaxation.

The BBN TC2000 is a shared memory multiprocessor capable of supporting up to 512 nodes. Each processing node consists of a Motorola 88100 processor, an 88200 cache unit, and several megabytes of memory. Each processor can access its own memory directly, and can access the memory on any other node through a $\log_8$-depth interconnection network (organized as an indirect binary N-cube composed of 8x8 crossbar switching nodes). Virtual circuit connections through the switching network are used to perform remote accesses. Many non-interfering virtual circuit connections (i.e., the sources and sinks of all connections are unique system wide) can be open across the switching network simultaneously. If collisions occur at a switch node, one transaction succeeds and all of the others are aborted, to be retried at a later time (in hardware) by the processors that initiated them. There is no hardware mechanism for ensuring coherence between cache units on different nodes. The 88200 cache unit allows any given memory segment to be configured with one of three caching policies: uncached, cached with write-through, or cached with copy back. Writes which go to main memory are performed so as to be sequentially consistent, i.e. they are slow because the process stalls until they complete. Transparent to the caching policy, the TC2000 architecture supports interleaved memory. In a page of interleaved memory, cache line size blocks are allocated in virtual address order across all processors in the machine in a round robin fashion. Interleaving can reduce potential contention for shared data.

The test programs all use the "Uniform System", a standard library of functions that support parallel programming on the TC2000. The tasking model supported by the Uniform System dedicates a set of processors to a program for its duration. Under the Uniform System, writable shared data must either be allocated uncached or the programmer must explicitly manage coherence with INV and UPD calls.

The sample codes used in our experiments were not optimized by hand in order to avoid prejudicing the test results. The simple sequential code with appropriate DO loops changed to parallel was used. Interleaving was used for all shared data. All caches were emptied before the start of each test. The times include the cost of faulting in the data initially, whatever evictions occur, and writing back the final results (except for the *no coherence* case, explained below).

The FSI method calls for additional hardware to invalidate cache contents in constant time. This hardware was not available on the TC2000. The FSI tests were run without the invalidate. The numerical results were wrong but the timing upper bounds real performance. We ran many of the tests again with the invalidate, which lower bounds performance because of software overheads, and found the two numbers to generally be quite close. The upper bound is reported in the graphs (as the more favorable to FSI).

The CKM method has been slighted somewhat because it was never designed to deal with cache line sizes of other than one word. On the TC2000, two 'double's fit in one cache line. Therefore, each fault in the CKM case causes 16 bytes instead of 8 to be read. None of the suggested CKM optimizations (§5.3) were applicable to these test cases.

The CTV method as presented so far does not rely on being able to understand array sections in any way. The code actually produced has many duplicate invalidations of the same locations. If array section analysis were available, CTV could be changed so as to invalidate the smallest (describable in vector notation) section that subsumed all others for which complete subscript information is available. This approach is referred to as CTV_{opt}, Optimistic CTV. This can over-invalidate and will not be better in all cases. What separates it from CTV is removing the overhead of unnecessary calls. For the tests cases in this paper, CTV_{opt} has an invalidation (but not update) pattern like FSI.

The three software methods are compared to the *no coherence* case. In this case, the program is run using a copy back caching strategy with no coherence provided in software. Read misses still occur to bring the data in the first time, but the final results are never written back and no values are communicated between processors (with the exception of values that are written out as a result of evictions and subsequently faulted in by another processor). The computed result is nonsense, but the performance in the no coherence case would seem to represent an upper bound on the performance that any coherence scheme, hardware or software, could hope to achieve. But for eviction anomalies, it is super-optimal because some coherence traffic is essential for the results to be correct.

The results are summarized in figures 5.8 through 5.12. Figure 5.7 is the legend for all of these graphs. Listed sizes are the number of rows in the principal matrix and not the total number of elements. Speedup for all figures, except 5.11, is versus one processor running in the no coherence case. For figure 5.11, speedup is versus the 20 processor uncached case for *each* problem size, i.e. the uncached case is defined to be a horizontal line. For that reason, figure 5.11 is not directly comparable to the others but it does more naturally represent the changing relation between the coherence strategies as problem sizes scale.

Each test point was run 30 times. A 95% confidence interval was calculated assuming a normal distribution. In most cases the interval was too small to even be visible on the graphs. This is not a confidence interval for an arbitrary run because overall system loading conditions affect timing through the memory interconnect. The data accurately reflects the relation between any two methods though, since all were run under similar loading conditions. These figures are not meant to show speedups over a well coded single processor version of the algorithm but only to show the relation of the different coherence methods.

The LU decomposition is a partial pivot blocked right-looking algorithm (§B.2). The blocking factor for all tests was 10. That is close to optimal for all measured test cases. The *no coherence* case produces numerous floating point exceptions due to both division by zero and overflow errors. Trapping these exceptions skews the results. To avoid this problem, all test cases for LU decomposition were run with a min operator in place of multiply and divide in at attempt to replace them with a nearly time-equivalent operator. This makes all test cases run about 3% slower but does not affect the relative performance. In particular, it does not affect the memory reference pattern. The pivot decisions were fixed to be the same for all cases (though the search for the best pivot still occurs). This did not measurably affect the running time.

LU decomposition is an O(n^3) time algorithm working on an O(n^3) matrix. Therefore each matrix element is being written (on average) O(n) times. Most of these writes could stay in cache. CTV takes advantage of this in its creation of a section to update thereby raising the CWE. The performance of FSI is degraded by doing these unnecessary writes. CKM simply has too much overhead to achieve good performance. FSI performs better than CTV on small matrices because CTV is paying higher overhead for system calls. For larger matrices, CTV performs better (figure 5.11).

Matrix Multiply (figure 5.8) responded similarly to LU decomposition but the advantage for CTV was even greater because all of the subscripts could be completely analyzed. In LU decomposition, such things as searching for the pivot then swapping whole rows based on it necessarily impact CTV's ability to do a good job.

Another advantage of CTV is that interleaving is not critical. FSI's maximum speedup is small when running with the default allocation of all memory on one node because of high contention. This is an artifact of the implementation of the "Uniform System" and can be fixed. But, CTV continues to perform well without worrying about this kind of tuning (figure 5.10).

The heat flow algorithm gives the FSI method its best chance to do better than CTV. The computational kernel of the algorithm uses a simple four point iterative relaxation in which the value computed for each point is needed in the next time step to compute the new value of each neighboring point. Lacking any knowledge about processor scheduling, it is necessary to update to main memory every write that occurs in the inner loop of this algorithm. The CWE is undefined since $W_t = W_o$ and cannot be changed due to the nature of the algorithm. Still, the advantage of handling a whole block at one time allowed CTV sometimes to do better than FSI and to be competitive in all cases. Without interleaved memory, CTV does noticeably but not dramatically better. Figure 5.12 shows the performance of the heat flow relaxation algorithm using each of the caching strategies.

We believe this data shows that the CTV method of aggregating coherence statements can reduce memory contention, amortize the inefficiencies of numerous system calls, and exploit write-back caches by relaxing the consistency model. Even in those situations where the intrinsic communication costs are high, CTV does not add an unacceptable amount of overhead. In contrast, FSI requires write-through. For machines that implement this without buffers, CTV can significantly improve performance. For machines with fast, buffered writes, FSI's overheads will be lower that CTV's. In some cases, CTV will have a better hit rate (chapter 9) and that will partially compensate for the overhead costs. But, in general, we expect fast, buffered writes will allow FSI to regain the execution time advantage but at the implementation cost of those buffers and special cache control. CTV is applicable without these additions and to already existing machines, such as the TC2000, that were designed without any special attention paid to coherence. If one is willing to pay the cost of special coherence hardware, far more effective strategies suggest themselves (sections 4.3, 4.5, and chapter 8).

Unfortunately, at present we cannot directly compare the results in any of our test cases to the performance of a hardware scheme. However, the performance in the *no coherence* case also represents an upper bound on the performance of any hardware based solution. Extrapolating from our measurements, the performance of CTV would appear to be comparable, on the TC2000, with any hardware-based strategy for each of the algorithms studied.

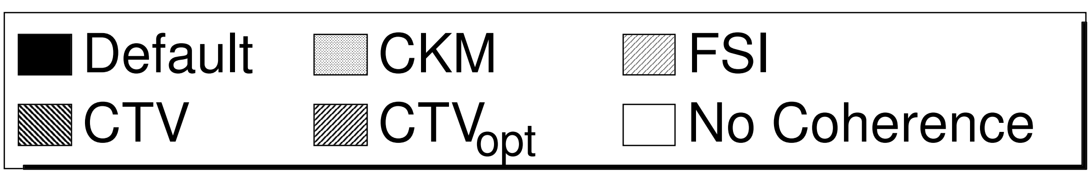


Figure 5.7: Legend for all CTV Graphs

Matrix Multiply

Size =50, Interleaved

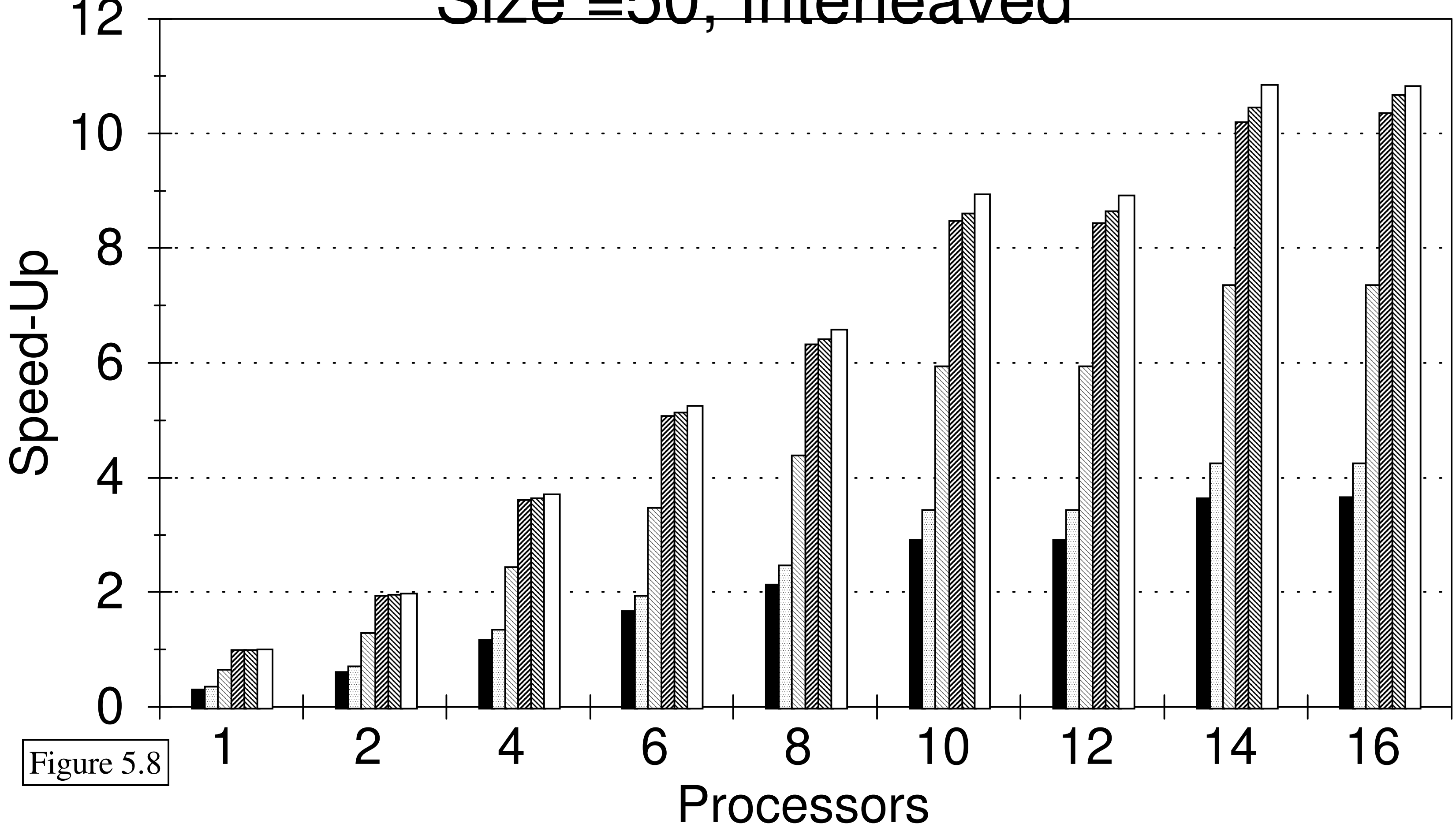


Figure 5.8

LU Decomposition

Size =200, Interleaved

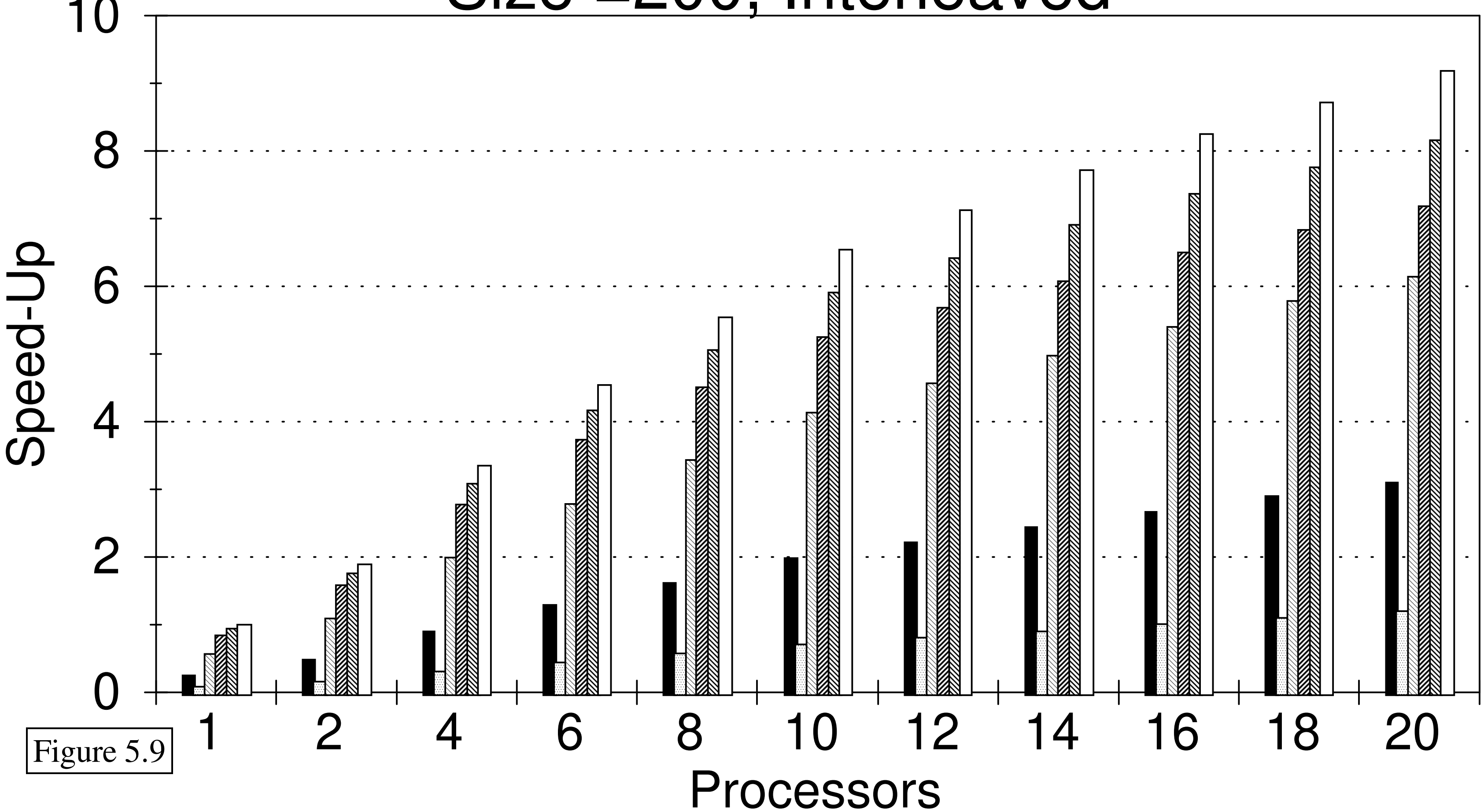


Figure 5.9

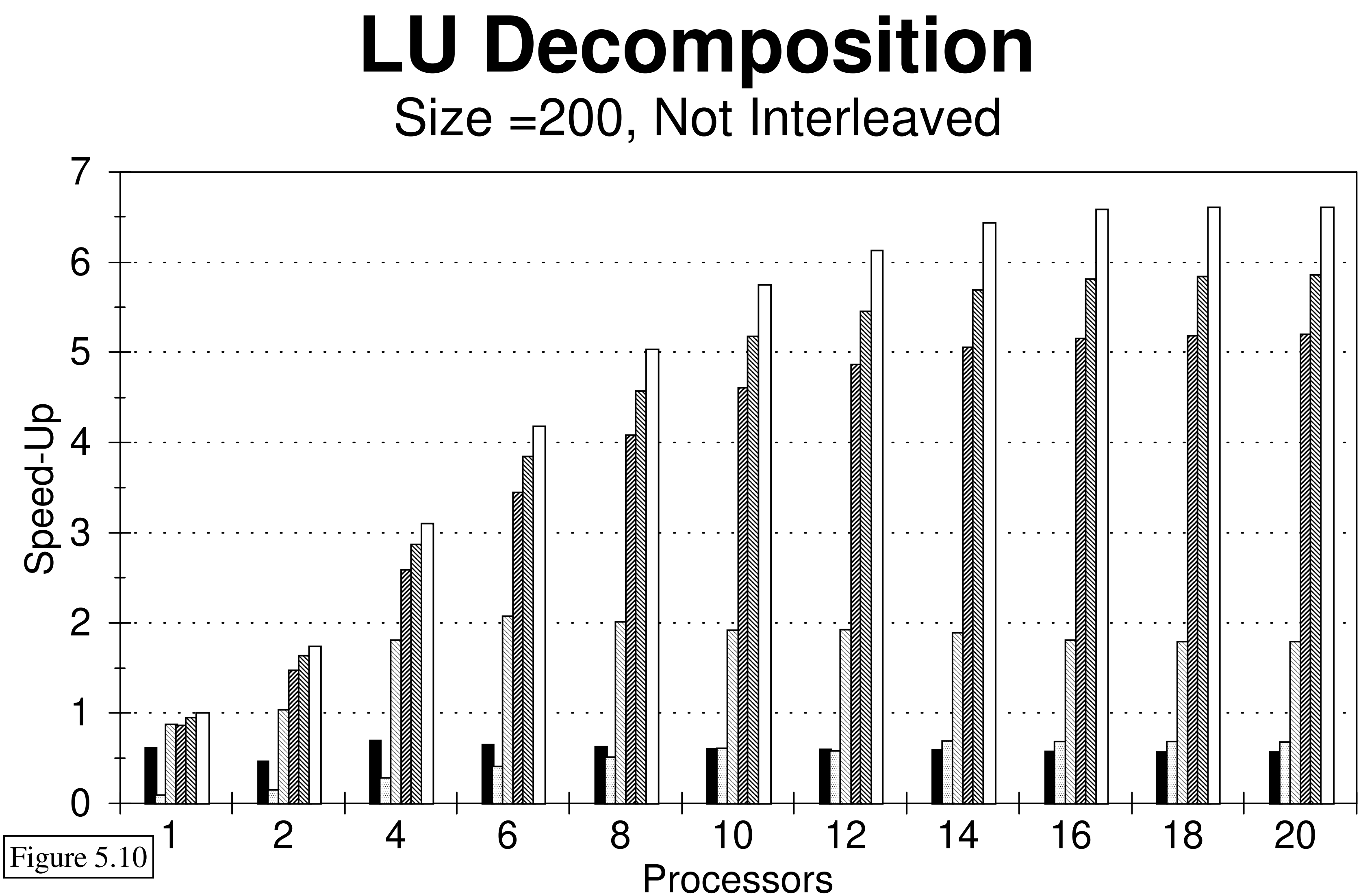

LU Decomposition
Size =200, Not Interleaved
Speed-Up
0
1
2
3
4
5
6
7
1
2
4
6
8
10
12
14
16
18
20
Processors

Figure 5.10

LU Decomposition

Interleaved, 20 Processors

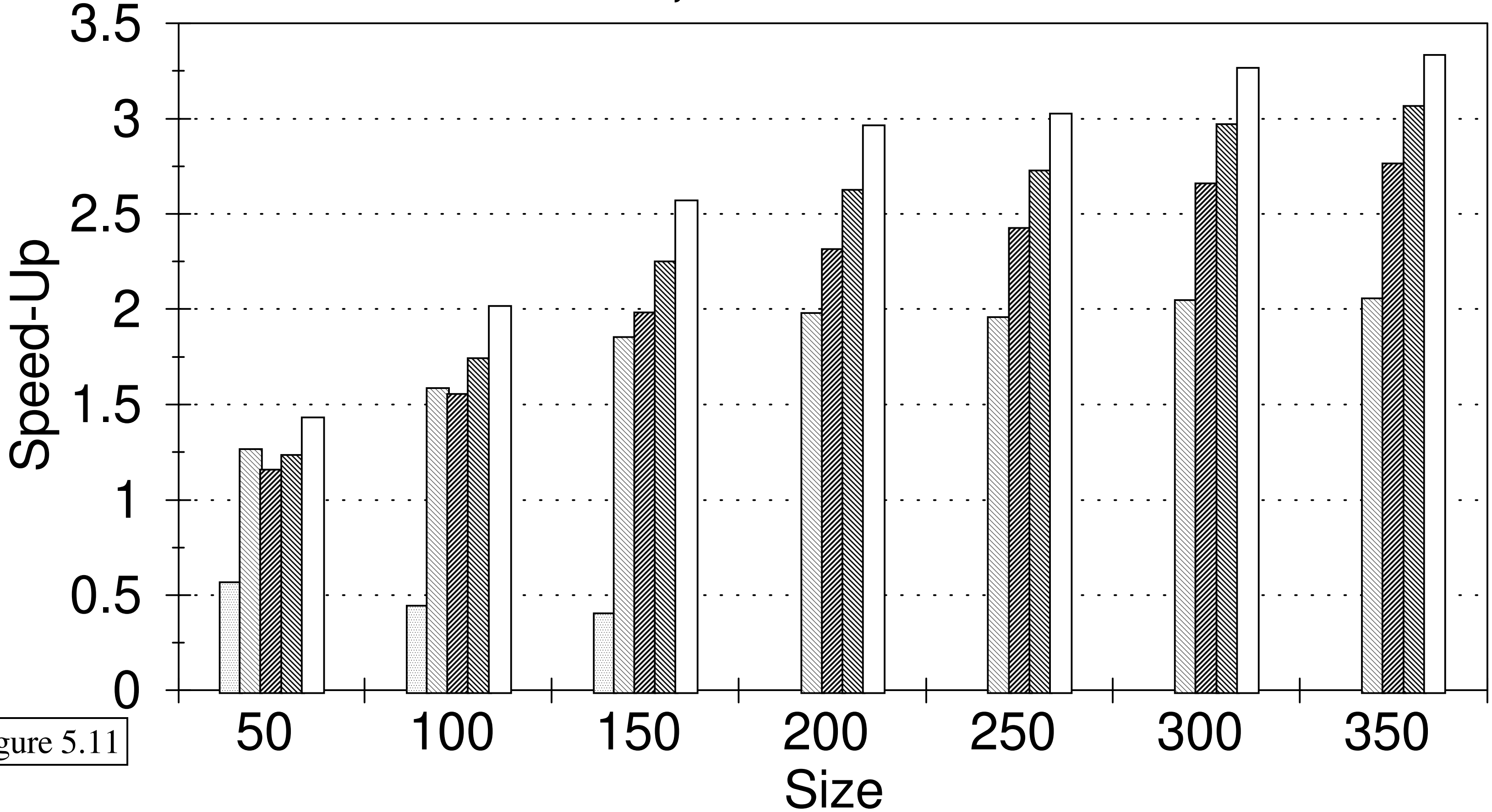


Figure 5.11

Heat Flow

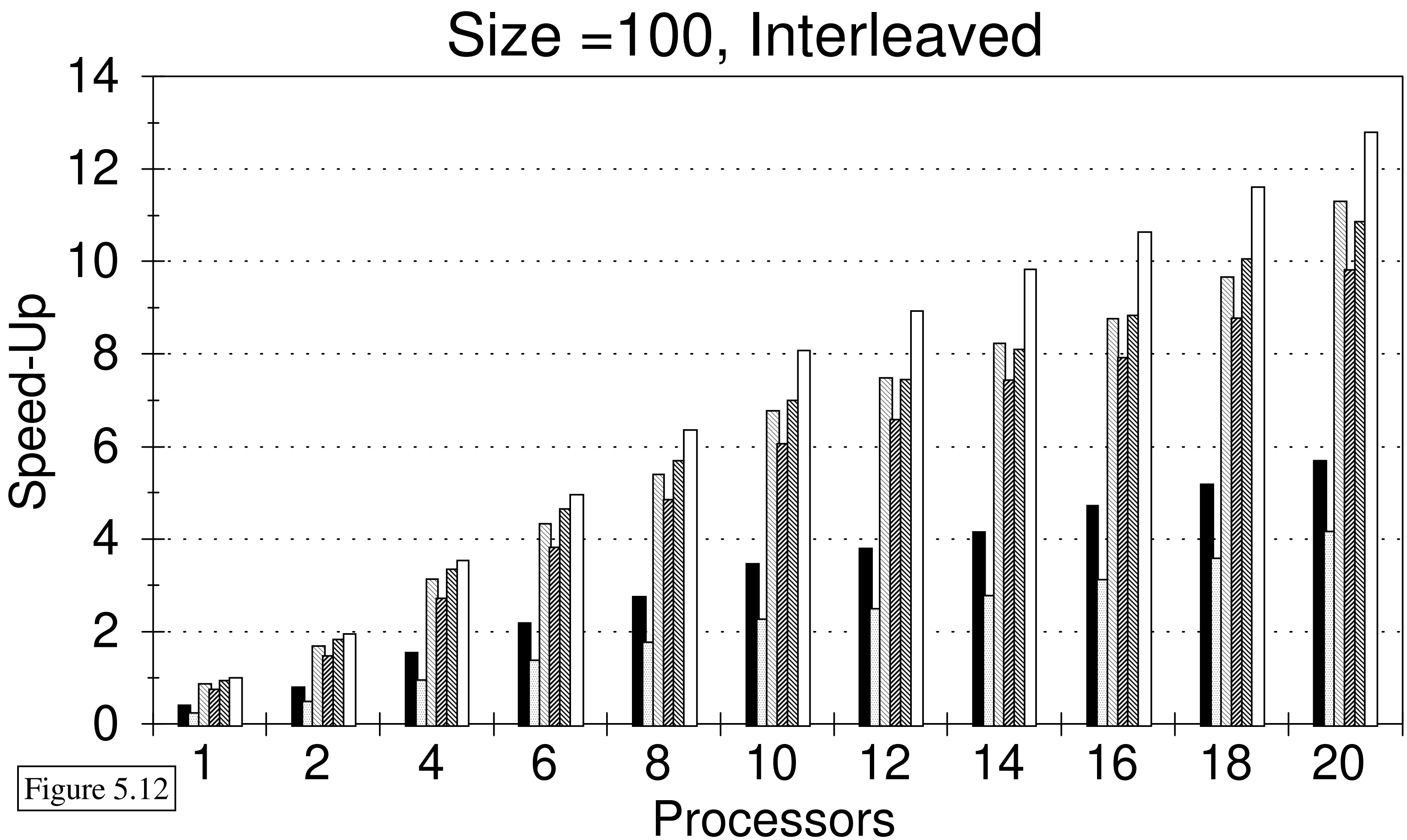


Figure 5.12

Chapter 6

CTV+ – A Static Inter-Epoch Approach

CTV takes an unnecessarily conservative view of possible ATs. It views any processor crossing dependence as a possible AT. It might seem that a static strategy can do no better than this. When a value is found in the cache at A_r (in an AT, (A_{ac},A_w,A_r)) there is no *a priori* way to know whether it got there from A_{ac} or A_w. The worst case is A_{ac} and thus an invalidate is needed. This is a necessary consequence of the CTV approach of never invalidating something until it is certain it will be accessed.

By accepting some over-invalidation, it is possible to do better than this. It is only necessary to invalidate an AT once, anywhere along the AT will do. For instance, if the invalidate follows A_w, there is no need for one between A_{ac} and A_w, and similarly for preceding A_w (definition 3.4).

Stated in isolation, this is a trivial observation. However, in the context of real code with multiple overlapping ATs, this has important applications. Consider:

```
PDO I
   =A¹(I)
END
PDO I
   A²(I)++
END
PDO I
   A³(I)++
END
PDO I
   =A⁴(I)
END
```

There are two relevant ATs here, α_1=(A^1,A^2,A^3) and α_2=(A^2,A^3,A^4). Invalidating for α_1 after A^2 also breaks α_2. Thus, one invalidate is sufficient. Reuse between the first and second epoch is preserved to the extent that they have similar schedules. This will often be the case. Reuse is also preserved between the third and fourth epochs. It is lost only between the second and third epochs. Thus, even a static strategy can preserve some *inter*-epoch reuse, 2/3 in this case.

CTV+ is an inter-epoch static strategy that uses this approach. It only places invalidates between epochs. This placement is based on a section analysis and not merely dependence analysis. It also utilizes special placement logic, beyond mere vectorization (as CTV used). Like CTV, it is suitable for machines without special coherence hardware.

CTV maintains exact subscript analysis by placing invalidates close enough to actual references that no locations are unnecessarily invalidated because of approximate analysis. This has two problems. First, intra-epoch reuse may be lost because the invalidate is necessarily trapped within a serial loop inside of a parallel loop. Second, there is additional overhead for each invalidate call when it may be possible to aggregate the calls. Even though this might lower the hit rate, it could be a good trade-off. CTV+ addresses both of these problems at the cost of including some data which is not stale at all.

The CTV_{opt} data (chapter 5) represents the case where all invalidates were moved to the beginning of the epoch and executed as a single call (for that particular test suite). Even though this approximates the data that actually needs invalidation it can improve the actual performance (e.g. figure 5.9). CTV+ extends this and, as a matter of design, has better hit rates and execution times than CTV_{opt}. We have present additional hit rate data for CTV+ after discussing TS1 and TS' (chapter 9) that show CTV+ to do significantly better than CTV.

The rest of this chapter presents the analysis and proofs of how to place inter-epoch invalidates effectively in a static strategy. First, we show that for sequential (inter-epoch) code, a simple greedy algorithm is optimal. It is simply: go through the nodes in order, greedily collecting as many as possible, and then place an invalidate only when necessary to preserve coherence. Several other special, but common, cases, e.g. simple fork-join, are analyzed. We do not know of a complete solution. This is unlikely to matter in practice though as we show in section 6.2.

6.1 Greedy Algorithm

The first step is to extend this observation to any set of ATs over sequential code. Here we assume a simple cost model that breaking an AT costs the same wherever it is broken, i.e. the next epoch will miss and subsequent ones will hit so that the miss cost is paid once somewhere. Write misses are assumed to have the same cost as read misses because the whole cache line is brought in. Maintaining coherence on ATs for different variables (or different sections) is a separable problem. Therefore, the naïve cost model is not far wrong. The number of misses expected to bring in valid data is the same for any invalidate point on a given AT. This logic still applies when multiple ATs for the save variable overlap because the size of the variable (or section) is the same for all of these ATs. This naïve model falls short only because the number of epochs between two accesses affects the probability that the data is still available for reuse (the greater the distance the higher the chance it has already been evicted).

With this cost model, there is very simple coherence strategy:

> Definition 6.1: Greedy Algorithm for optimal static invalidate placement on sequential inter-epoch control flow:
>
> Process each epoch in order. Add an invalidate only when failing to do so would not preserve coherence.

This is optimal with respect to the cost model. The greedy algorithm is O(n). It places no constraints on the topology of the ATs (only on the control-flow graph). The ATs can span an arbitrary number of epochs and overlap each other in any fashion. We show later (§6.2) that this level of generality is not necessary for understanding most real code, but the formal proof here stands as a solid check against later, more intuitive, descriptions.

For the proof, a maximal optimal solution is defined so as to be the right thing. The greedy solution is then shown to be the same solution. We know that it is a solution because worst case, invalidating before every node preserves coherence. The inductive hypothesis is that the greedy algorithm is optimal for a program with n-1 nodes, the induction then shows it must also be optimal for n nodes. There are two cases to consider: First, an optimal solution for n nodes is larger than n-1 nodes; second, an optimal solution for n nodes is the same size as one for n-1 nodes (but not necessarily the same solution). In each case, it can be shown the greedy solution breaks every access triple. Access triples that involve nodes $<n$ are easily handled by the inductive hypothesis because the greedy algorithm never rearranges previous solutions. It is then left to be shown that any ATs which involve node n are properly handled. In general, each step relies on the previous one as one of its reasons, but these are not listed to avoid clutter.

1. Definitions

Label	Step	Comment
	Epoch = {i | 1≤ i ≤N}	a set of epochs
	$u,v \in V$	program variables
	$v_r \subseteq$ Epochs	nodes that read v
	$v_w \subseteq$ Epochs	nodes that write v
	$v=v_r \cup v_w$	nodes that reference v, $v_r \cap v_w$ is not necessarily null
	$\Phi = \{(\alpha_{ac},\alpha_w,\alpha_r) \in \text{Epoch} \times \text{Epoch} \times \text{Epoch} \mid \alpha_{ac} < \alpha_w < \alpha_r\}$	All possible access triples (ATs)
	$\beta = (\beta_{ac}, \beta_{re}) \in \text{Epoch} \times \text{Epoch} \mid \beta_{ac} < \beta_{re}$	A given reuse arc (RA) [thm. 3.1]
	$v_{at}=\{\alpha \in \Phi \mid \alpha_{ac} \in v \wedge \alpha_w \in v_w \wedge \alpha_r \in v_r\}$	All access triples (ATs) for a given variable
	$\overline{\alpha} = \{i \mid \alpha_{ac} \le i < \alpha_r\}$	An AT span (half-open)
	$v_{ra}=\{\beta \mid \beta_{ac}, \beta_{re} \in v\}$	All reuse arcs (RAs) for a given variable
	[After this point, all sets are understood to be for a particular v]	
D1	$\Phi^n=\{\alpha \in v_{at} \mid \alpha_r \le n\}$	ATs before a certain node
D2	$\mathbf{T}^n=\{T^n \subset \text{Epoch} \mid \forall \alpha \in \Phi^n\ \exists k \in \mathrm{T}^n \mid k \in \overline{\alpha}\}$	The set of solutions where a solution is a set of invalidates that cut every AT
D3	$\mathbf{S}^n=\{x \in \mathbf{T}^n \mid \neg\exists\ y \in \mathbf{T}^n \mid \lVert y \rVert < \lVert x \rVert\}$	Optimal solutions, those with the fewest number of invalidates
D4	$top^n = \max\limits_{x \in \mathbf{S}^n}(\max(x))$	The highest element of any optimal solution
D5	$\mathbf{S}^n_{\mathfrak{m}} = \{x \in \mathbf{S}^n \mid \max(x) = top^n\}$	The set of optimal solutions containing the highest element
	Let $S^n_{\mathfrak{m}} = \mathbf{S}^n_{\mathfrak{m}}$	The first such element in any canonical order

2. The Greedy Algorithm

	Step	Reason	Comment
G0	$G^1 = \{\}$		
G1	if $G^{n-1} \in \mathbf{T}^n$ then $G^n=G^{n-1}$		if previous solution solves size n, then keep previous solution
G2	else $G^n = G^{n-1} \cup \{n\}$		else, add the most recent node
Thm	$G^n \in \mathbf{S}^n$		Greedy is Optimal
IH	$G^{n-1} \in \mathbf{S}^{n-1}_{\mathfrak{m}}$		Inductive Hypothesis

Base	$\Phi^1=\{\}$, $G^1=\{\},\{\} \in \mathbf{S}^n_{\mathbb{m}}$	D1,D2, G0	a single node is trivially coherent since there are no ATs

a) Case 1 of Induction

CH	$\lVert S^n_{\mathbb{m}} \rVert = \lVert S^{n-1}_{\mathbb{m}} \rVert + 1$		case hypothesis
(1)	let $t = S^{n-1}_{\mathbb{m}} \cup \{n\}$		a possible solution
	$\forall \alpha \in \Phi^n$		for every AT,
	if $\alpha_r < n$, $\exists\, k \in S^{n-1}_{\mathbb{m}} \mid k \in \overline{\alpha}$	D2	$S^{n-1}_{\mathbb{m}}$ is valid for all nodes $< n$ by IH
	else $n \in \overline{\alpha}$	D1	n itself makes α coherent
	$t \in \mathbf{T}^n$	D2	t is a solution, there is an invalidate for all ATs
	$\lVert t \rVert = \lVert S^{n-1}_{\mathbb{m}} \cup \{n\} \rVert$	(1)	is t an optimal solution?
	$= \lVert S^{n-1}_{\mathbb{m}} \rVert + \lVert \{n\} \rVert$		$\{n\}$ and $S^{n-1}_{\mathbb{m}}$ are disjoint
	$= \lVert S^{n-1}_{\mathbb{m}} \rVert + 1$		
(2)	$= \lVert S^n_{\mathbb{m}} \rVert$	CH	it has the right size
(3)	$\max(t) = n \geq top^n$	D4	it has the correct top element
(4)	$t \in \mathbf{S}^n_{\mathbb{m}}$	(2), (3), D5	t is an optimal solution
	$\lVert G^{n-1} \rVert < \lVert S^n_{\mathbb{m}} \rVert$	CH, IH	the previous greedy solution is smaller than an optimal solution
	$G^{n-1} \notin \mathbf{T}^n$	D2, D3	and is therefore not a solution here
(5)	$G^n = G^{n-1} \cup \{n\}$	G2	$\{n\}$ must be added by the anti-greedy algorithm
(6)	$\lVert G^n \rVert = \lVert t \rVert$	(1), (5), IH	same size as t
(7)	$\max(G^n) = \max(t)$	(1), (5)	same max element as t
	$G^n \in \mathbf{S}^n_{\mathbb{m}}$	(4), (6), (7), D5	by (4) and the definition of an optimal solution, the constructed G is optimal

b) Case 2 of Induction

CH	$\| S^{n}_{\mathbb{m}} \| = \| S^{n-1}_{\mathbb{m}} \|$		case hypothesis
	$S^{n}_{\mathbb{m}} \in \mathbf{T}^{n\text{-}1}$	D2	an optimal solution must also solve the previous case
	$S^{n}_{\mathbb{m}} \in \mathbf{S}^{n-1}$	D3, CH	it must also be optimal for the previous case because of its size
(1)	$\max(S^{n}_{\mathbb{m}}) \le top^{n\text{-}1}$	D4	therefore, its max is constrained
(2)	$\forall \alpha \in \Phi^{n}, \alpha_{ac} < top^{n\text{-}1}$	D2, D5	true by definition of *top*[ED28]
(3)	$\max(G^{n-1}) = top^{n-1}$	IH	
	$\forall (i, j) \in A^{n}$		for all access triples, either they are handled
	if $\alpha_r < n$, $\exists\, k \in G^{n\text{-}1} \mid k \in \overline{\alpha}$	D2	by the IH greedy solution
	else $\alpha_r = n$, $\max(G^{n\text{-}1}) \in \overline{\alpha}$	(2), (3)	or by the special properties of top just shown
	$G^{n\text{-}1} \in \mathbf{T}^{n}$	D2	therefore, the previous greedy solution solves this case
(4)	$G^{n} = G^{n-1}$	G1	the construction of the anti-greedy solution
(5)	$\| G^{n} \| = \| G^{n\text{-}1} \| =$ $\| S^{n-1}_{\mathbb{m}} \| = \| S^{n}_{\mathbb{m}} \|$	IH, CH	the new anti-greedy solution has the right size
	$\max(G^{n}) = \max(G^{n\text{-}1}) =$ $\max(S^{n-1}_{\mathbb{m}}) \ge \max(S^{n}_{\mathbb{m}})$	(4), IH, (1)	and it is not smaller than the required max
	$\max(G^{n}) \le \max(S^{n}_{\mathbb{m}})$	D5, (5)	by definition of the maximal optimal solution, this must be true, so G has the right max
	$G^{n} \in \mathbf{S}^{n}_{\mathbb{m}}$	D5	
	Q.E.D.		

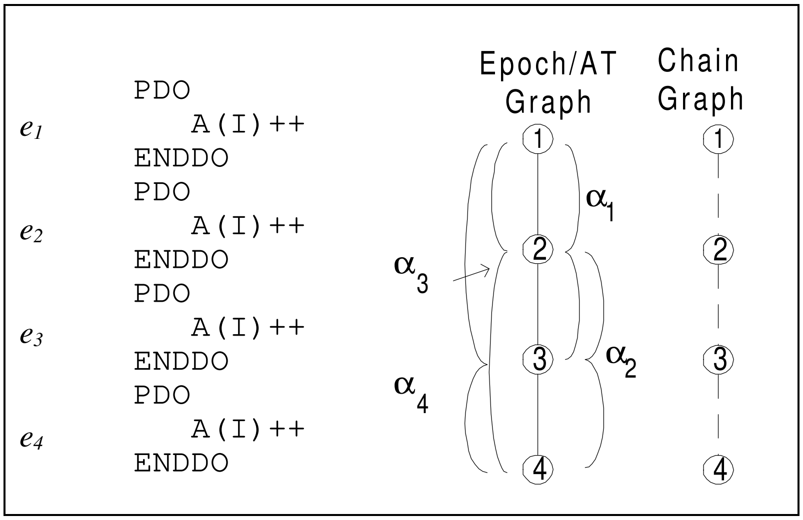


Figure 6.1: Chain Example

6.1.1 Superset ATs

An implication of any static strategy is that an AT which supersets another can be ignored for the purposes of finding invalidate positions that preserve correctness. For instance, in figure 6.1, there are four ATs, α_1 through α_4. Any invalidate which cuts α_1 also cuts α_3 and α_4. Therefore, the latter two can be ignored.

That correctness is guaranteed by ignoring an AT which supersets is not the same as optimality being assured by it. This reflects on the cost model assumption that breaking an AT costs the same wherever it is broken. This fails to be true when reuse at an epoch can come from different prior epochs. Under some probabilistic scheduling assumptions, there is more reuse in this case than for a single prior epoch because only a certain fraction of each is available as inter-epoch reuse.

6.2 Chains

A common case of multiple ATs is where every node reads and writes a particular variable. In this case, it is convenient to summarize the ATs as a *chain* of nodes with the understanding that any three adjacent nodes form an AT (e.g. figure 6.1). Nodes that neither read nor write a variable are ignored and do not interrupt chains. Instead of showing the ATs to the side, the chain directly connects the nodes. The definition of chain is extended slightly to allow the possibility of the last epoch node being only a read, the penultimate being only a write, and the first being any reference. A chain is a sequence of nodes of the form (r|w|rw)(rw)*(w|rw)(r|rw).

Each AT spans exactly three adjacent nodes in a chain. Longer ones can be ignored as shown in the previous section (§6.1.1). An important aspect of a chain is that represents the only fashion in which ATs which do not superset can overlap each other:

Theorem 6.2: No non-supersetting AT crosses a read-only epoch.

Proof:
Assume such an AT, α, exists.
Let v_r = the read-only epoch
$\alpha_{ac} < v_r < \alpha_r$ – since it crosses
Case 1: $\alpha_w = v_r$ – cannot be, v_r is a read-only epoch
Case 2: $\alpha_w < v_r$, $(\alpha_{ac},\alpha_w,v_r)$ demonstrates α is a supersetting AT
Case 3: $\alpha_w > v_r$, (v_r,α_w,α_r) demonstrates α is a supersetting AT

A similar property holds for write-only epochs:

Theorem 6.3: No crossing AT for writes

Let v_w = the write-only epoch
No non-supersetting AT crosses α'_{ac} where $(\alpha'_{ac},v_w,\alpha'_r)$ is a non-supersetting AT and $v_w - \alpha'_{ac}$ is minimal.
Proof:
Assume such an AT, α, exists.
$\alpha_{ac} < \alpha'_{ac} < \alpha_r$ – meaning of crossing
Case 1: $\alpha_r = v_w$ – cannot be, v_w is a write-only epoch
Case 2: $\alpha_r < v_w$
(α_r,v_w,α'_r) demonstrates α' is a supersetting AT
Case 3: $\alpha_r > v_w$
$(\alpha'_{ac},v_w,\alpha_r)$ demonstrates α is a supersetting AT

These two theorems taken together demonstrate that any node which is either a read-only or write-only node splits the ATs into two sets, one before and one after the solo node, such that no AT connects to both sets. Thus, the invalidate pattern on one set can have no direct effect on the other. The topology of all graphs then becomes a series of independent chains. The greedy algorithm becomes a trivial observation in light of this. For reference, we restate this as an observation:

Observation 6.4: Epoch graphs are a set of chains

```
     PDO
e1     A(I)=
     ENDDO
     PDO
e2     A(I)++
     ENDDO
     IF (c)
        PDO
e3        A(I)=
        ENDDO
     ELSE
        PDO
e4        A(I)++
        ENDDO
     ENDIF
     PDO
e5     A(I)++
     ENDDO
     PDO
       =A(I)
e6   ENDDO
```

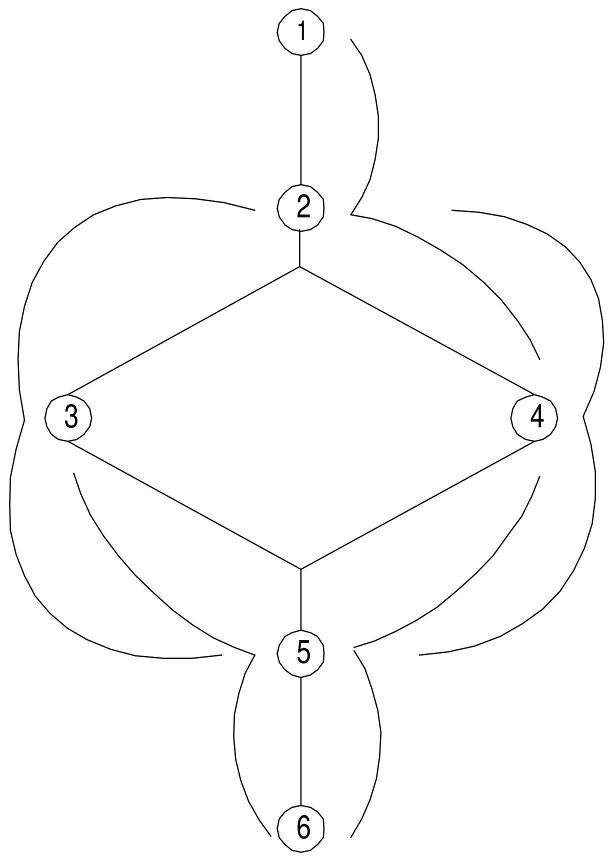


Figure 6.2: Example ATs on fork-join graph

6.3 Fork-Join control flow

6.3.1 Trees

Observation 6.4 is still applicable for DAG inter-epoch control flow. The implications are not as simple though. Figure 6.2 is an example where the chain only goes around one branch. Even though the ATs around the left branch cross each other and would seem to represent a continuation of the chain, this is not actually so because any invalidate which follows e_2 could be moved to both the left branch and the right branch. This would not in any way affect execution time or hit rate. After such a move, there is no AT crossing the left branch AT with the write-only node, (e_2, e_3, e_5), as per theorem 6.3. The chain graph become a 'T' after this break (figure 6.3). Similar comments apply for the read-only case.

The important aspect of this graph is that the main chain, epochs 1, 2, 4, 5, & 6 can be solved with the greedy algorithm independently of e_3. Whatever conclusion is found there then forces the answer for e_3. If the greedy algorithm is applied to any chain tree, it will generate an optimal solution, for the CTV+ cost model, for the same reason it does in the linear case. However, this is not an adequate model. Losing reuse on a branch of an `IF` is only half as expensive on average as losing reuse before or after the `IF`. Incorporating this into the model leaves the greedy algorithm sub-optimal.

To address this, consider a simple chain again (figure 6.1) and how far wrong one can go. Clearly, there must be invalidate at least every other epoch to cut every AT and thus to preserve correctness. In this single chain case, there is no need for more than that. The only question remaining is whether to put the first invalidate after e_1 or e_2. In this case, putting it after e_1 would then require another one after e_3. That would cost one more than the optimal. This is true in general, either an extra invalidate will be needed at the end of a chain or not. Thus, the worst case penalty of a mistake in the initial invalidate placement is 1 for a single linear chain. Omitting proof, we state this as an observation:

> Hypothesis 6.5: Optimal linear chain invalidation
>
> The optimal invalidation pattern for a linear chain is a pattern with an invalidate every other epoch. Any pattern invalidating every other epoch either has the optimal cost or the optimal cost + 1 (depending on which epoch is the first to be invalidated).

In a tree, consider the chain along each path to be a separate linear chain. For instance, there are two such chains in figure 6.3, c_1=e_1, e_2, e_4, e_5, e_6 and c_2=e_3, e_5, e_6. Each chain now has cost 1/2 because part of each travels one branch of the `IF`. The cost of an invalidate between e_5 and e_6 is 1 regardless of which chain it is on behalf of, but the cost is amortized partially for each chain. Now, solve each chain independently. The only remaining question is: do they agree or conflict when merged? That is, does one invalidate after e_5 and the other does not?

The important point to this is that not reconciling them cannot be better than the optimal solution (it may be equal though). The worst case cost of starting in the wrong epoch is 1/2 for either linear chain (Hypothesis 6.5). The cost of failing to reconcile is also 1/2 for whichever chain is cut unnecessarily at e_5. For instance, if c_2 has an invalidate after e_3 and c_1 has an invalidate after e_2 and e_5 (both optimal in isolation) then the invalidate at e_5 also cuts c_2 and costs reuse unnecessarily when the left branch is taken. It cannot be a worse solution to let c_2 be invalidated at e_5 then handle the rest of it as need be. We generalize this as:

> Hypothesis 6.6: Optimal chain tree invalidation
>
> An optimal invalidation pattern on a chain tree has invalidates every other epoch along any linear chain (that exists along a valid control flow path).

The implication is that all merged linear chains agree. This leads to an optimal algorithm (under this simple cost model) for placing invalidates: try both possible locations for the first invalidate in the first chain and let everything else fall in place by the greedy algorithm.

6.3.2 DAGs

How to properly handle chains that form DAGs remains a matter of future research. Even whether or not the general case has a polynomial time solution remains an open problem. We present results for some simple, common cases. First, chains that form trees can be handled later as previously discussed. The rest of this section assumes that every node in question is a read-write node and that chains connect all of the paths in question. We use the following notation for discussing control flow paths:

- n – a chain n nodes long
- p,q – paths
- pq – a sequential composition of two paths
- $<p,q>$ – an `IF` with path p as the left branch and path q as the right branch
- C_p – the cost along some path (number of invalidates)
- I – location of invalidate
- D_p – a name for the path/placement of invalidates on the path

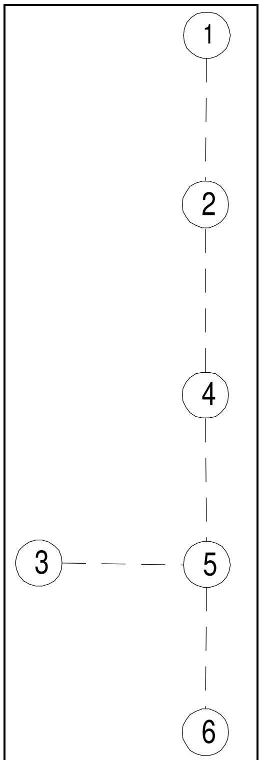


Figure 6.3: Chain graph for figure 6.2.

For instance, a chain of 4 nodes can be handled with a single invalidate in the middle, i.e. C_4=1 by virtue of D_{2I2}. Hypothesis 6.5 shows more generally that:

$$C_n = \left\lfloor \frac{n-1}{2} \right\rfloor$$

For the purposes of the section, it is useful to view this as $C_{n-2}=C_n-1$. This means that for examining interactions, the only interesting chains are those of length 0, 1, or 2, so long as the interaction pattern is not changed. For instance, $C_{2\langle 2,2\rangle,1}=C_{2\langle 2,4\rangle,1}-1/2$ (the right branch is only worth 1/2). This also specifies the placement of the invalidate by maintaining the existing 'every other' pattern wherever a pair of nodes is inserted. Attempting to use $C_0=C_2-1$ is not safe because it could eliminate the last two epochs on a branch and allow an AT to reach a different chain than before. For instance $D_{1\langle 2,1\rangle 2}$ is:

```
PDO I
   A(I)++         - e1
ENDDO
IF (c)
   PDO I
      A(I)++
   ENDDO
   PDO I
      A(I)++
   ENDDO
ELSE
   PDO I
      A(I)++
   ENDDO
ENDIF
PDO I
   A(I)++         - e5
ENDDO
PDO I
   A(I)++         - e6
ENDDO
```

If evaluated with a reduction on the *true* branch this would produce (e_1,e_5,e_6) as an AT. Before the reduction, no AT went from e_1 to e_6. That would change the topology and not merely the cost. Reducing 3 nodes to 1 is safe though. Even though one AT may reach into a new chain, it necessarily replaces a deleted one which already did the same. This does not change the topology.

The value of this reduction is that any particular DAG can have its optimal invalidation pattern found based just on its topology and not the number of nodes in it. Taking the simplest case, $D_{t<l,r>b}$, it is only necessary to examine 3^4 possibilities since all cases can be reduced to t,l,r,b ≤ 2. Removing symmetries and degenerate chain cases (e.g. t=0) leaves only 15 possibilities. Exhaustively searching them leads to the following:

$$C_{t<l,r>b}=\frac{1}{2}\left(\left\lfloor\frac{t+l+b-1}{2}\right\rfloor+\left\lfloor\frac{t+r+b-1}{2}\right\rfloor\right)$$

This is an interesting result because it is the same as if each possible path through the DAG were considered as an independent problem and then weighted by its probability of execution, i.e. $C_{t<l,r>b}=1/2(C_{tlb}+C_{trb})$. Attempting to solve each path independently for actual invalidate placement will *not* produce this cost outcome though. For $D_{t<l,r>b}$ a fast strategy is to solve an even path, if any, first, then finish solving the odd path. That always produces the minimal cost. The intuitive reason is that moving an invalidate one node on an odd path does not change anything, e.g. $C_{1I2}=C_{2I1}=1$, while there is only one good choice on an even path, e.g. $C_{2I2}=1$, so even paths are more critical and need to be handled first.

These observations do not generalize, at least not directly. For instance, the all-paths cost for $C_{0(1,0)2(1,0)0}=0.75$, but the actual minimum is 1.0. Similarly, always following the even path is not always the right strategy.

More special case results can be generated, but before pushing this further, several issues need to be addressed:

- How often does complex inter-epoch control flow occur in real programs?
- How far wrong is the cost model of ignoring supersetting ATs?
- To what extent is actual data volume a more important criteria?
- Should invalidates by parameterized by recording execution path histories?

These all remain as matters for future research. The simple rules presented here are sufficient to handle all programs for the test suite used in this thesis (chapter 9). It would be a rare program which could profit from a more detailed cost formulation of the control flow.

Chapter 7

TS1 – An Ideal Dynamic Local Strategy

7.1 Introduction

TS1 is an ideal local strategy that can utilize all available resolution of the dependence analysis. It does not depend on any values being known before the essential write of an AT which allows for the widest scope of symbolic dependence analysis. No local coherence strategy can ever achieve a better hit rate (for a given resolution of dependence analysis). It does however require a complex invalidate instruction with substantial execution time overhead.

For additional storage requirements, TS1 requires only a single *epoch* bit per cache line (beyond the *valid* bit which is already present) and no extra instruction bits. For extra logic, it needs a mechanism to invalidate an address range of cache lines, and the ability to simultaneously copy all *epoch* bits to their corresponding *valid* bits. The compiler algorithm is essentially to insert invalidates that invalidate everything at the end of an epoch that *appears* (to the compiler) to have been written in the epoch.

This improves on the best previous strategy, TS, with lower bit costs and higher hit rates (for compiler analysis at a finer granularity than whole array). However, the implementation of the invalidate can require substantially more hardware and impose its own run-time overhead. We consider some of the options below.

Improvement in actual execution time depends upon the distance to main memory, the particular program, the execution environment, and the efficiency of the available invalidate instruction. Where the inter-epoch reuse in a program occurs between compiler recognizable array sections, and not between whole arrays, TS1 gains a hit rate advantage over TS. As the amount of data per processor grows this translates into a performance benefit for TS1 because the miss cost of the hit rate improvement is greater than the loss due to the invalidate overhead. For those cases where using the finer resolution of section analysis does not produce any performance benefit for TS1, it can sacrifice the extra granularity and return to invalidating whole arrays as does TS. This saves most of the invalidate overhead but leaves it with the same hit rate as TS and some remaining overhead for invalidating whole arrays.

The implementation of the invalidate is orthogonal to TS1 *per se*. We present the strategy on the assumption that the invalidate exists. Fast ones are necessary to make TS1 efficient though. As well as presenting TS1 in the abstract, we consider two different possible invalidate implementations as used by previous authors in this area [1,47]. Regardless of whether a cost-effective implementation exists for either of these, the real value of TS1 is that it stands as an instance of an ideal local strategy against which the hit-rates (but not necessarily performance) of other local strategies can be compared. Any gap that remains is one which could, in principle, be closed.

It also addresses, though far from settles, the question of how well local strategies compare to global strategies. If the hit-rate gap between TS1 and global strategies is large, it is evidence in the negative that no local strategy can ever be adequate. In our data based on regular and readily analyzable scientific Fortran programs, this gap was usually < 1% and often exactly 0%. If this holds in general, it argues that local strategies can be effective in place of global strategies.

This chapter presents the details of a TS1 implementation. It presents some hit rate data and then discusses the execution implications based on varying miss costs. Finally, the additional issues of cache line size, `DOACROSS`, and critical sections are considered. More extensive hit rate data is provided in chapter 9 after TS' has been discussed.

7.2 Implementation

7.2.1 Hardware support

TS1 requires a *valid* bit per cache line and an additional bit, the *epoch* bit. In TS1, caches set the *epoch* bit on any reference to that line (read, write, hit, or miss). At the end of a given epoch, e_i, a special instruction resets the *epoch* bit for every line in the cache. We assume that the cache implementation can do this in O(1) time by having every cache line respond in parallel. Since all of the *epoch* bits were reset on entry to epoch e_i from the end of epoch e_{i-1}, the *epoch* bit reflects which cache lines have been accessed during epoch e_i. This much is the same mechanism as has been previously proposed for the FSI strategy [14] and serves a similar semantic purpose, recording current epoch accesses. However, TS1 puts this information to better use.

By the assumed semantics of `DOALL` loops (§3.1.2), any cache line with its *epoch* bit set in epoch e_i can be left in cache for epoch e_{i+1} without causing a stale access. We use this observation in defining a special *invalidate* that operates optimistically. When a particular cache line is the object of an invalidate, it is actually invalidated only if the *epoch* bit is reset, otherwise it remains valid and in cache, i.e. the invalidate copies the *epoch* bit to the *valid* bit.

7.2.2 Implementation of the Invalidate

There are several choices for the actual implementation of the invalidate that trade-off hardware cost for run time efficiency.

A slow but inexpensive implementation would be to have a low level invalidate instruction which could invalidate either a particular line or a particular page. The high level invalidate would then loop over the proper range of pages and lines. Even though this would take O(|section|), acceptable performance could still be achieved. We examined the efficiency of this kind of invalidate in the context of CTV (chapter 5).

A faster, but more complex, invalidate could work by using a bit mask to determine which addresses to invalidate. With only '=' comparators and no extra storage, a contiguous section could be invalidated in O(log(|*section*|)) time. Special layouts and strides could reduce this further. This is similar to what PEI does.

Other authors have proposed O(1) time invalidation implementations which work by accessing cache row and column addresses [1].

7.2.3 Software support

To determine what to invalidate, TS1 uses compile time analysis to determine what is written for each epoch. The compiler makes its best estimate that is sure to include every address actually written. The main task of this analysis is to determine which parts of shared arrays are written. A naive analysis could simply note which arrays appear on the left hand side of an assignment and then conclude that every element of any such array is modified. More sophisticated analysis could try to determine which sections of arrays are actually modified. For every section (or whole array) that is modified, the compiler inserts an invalidate for that range of addresses at the end of the epoch being analyzed. Since schedules are not known, the same set of invalidates is used for every processor.

At run time, for each epoch, some accesses occur, setting the *epoch* bits, then the invalidates are executed as the next to last instruction, removing soon to be stale values, and

		Shared	
		Valid Bit	Epoch Bit
State:	Fresh	1	1
	Valid	1	0
	Stale	0	0
Action:	Reference	1	1
	Invalidate	Epoch bit	-
	End of Epoch:	-	0
Effect on Reference:	Hit	1	
	Miss	0	

Table 7.1: TS1 Summary

finally all of the *epoch* bits are reset in preparation for the next epoch. This is a direct implementation of the local schedule theorem (theorem 3.5). Table 7.1 summarizes these rules.

To see that it is correct, consider a given AT (definition 3.1), $\alpha=(\alpha_{ac}[p_i],\alpha_w[p_j],\alpha_r)$. If $i=j$, then α_w evicts α_{ac} from cache. α_{ac} cannot now be referenced as stale. α_w possibly begins a new AT for which the essential write occurs later. If $i\neq j$, the epoch bit will not be set on p_i after α_w and the invalidate corresponding to α_w will remove $\alpha_{ac}$from cache. This fulfills the definition of coherence (definition 3.4). Given correctness and the local schedule theorem, TS1 is an ideal local strategy (definition 3.7).

If compiler analysis were perfect, TS1 would have the same hit rate as a global scheme. That is, the coherence strategy *per se* sacrifices nothing. The conservative assumptions that must be made at compile time cause some reuse to be missed. This is the loss that any local scheme must suffer.

The nature of the TS1 implementation has some impact on the sort of analysis the compiler can perform. Invalidates follow their corresponding writes. Nothing need be determined before the write occurs. There are many instances where this delay allows symbolic analysis to propagate a symbolic value into the invalidate without knowing its exact value (or even determining its exact relation to some other value). If decisions had to be made before the write such symbolic values would often (though not necessarily) be unavailable. An example appears later in conjunction with the opposite scenario (§8.2.1).

7.2.4 Contrast Between TS1 and Previous Strategies

TS1 and TS are both ideal local strategies at the level of compiler-analysis granularity they operate on. Both move cache lines through the same essential three states (figure 3.4). They achieve this effect in two opposite ways. TS works by updating the array clock on the processor and letting the cache state be determined *implicitly* by the relation between its time stamp and the array clock. The difficulty with this is since the cache line is not updated until it is referenced again (i.e. no invalidate touches it), the time stamp must remain valid with respect to the array clock indefinitely. This necessitates extra bits to count epochs. TS is a lazy strategy. TS1 takes an aggressive approach and records the relevant state directly in the cache line by using the *valid* and *epoch* bits. Because of this it only needs two bits per cache line.

There is a perhaps more subtle problem with the lazy TS approach. Not only must the time stamps in the cache lines stay valid but there must be a clock to compare them to. At first this does not seem to be a problem, but it becomes an issue if one were to try and extend TS to handle sections. There would need to be a clock for every section. This could quickly become a significant cost (the question of choosing the right clock becomes considerably more difficult than for whole array analysis). The aggressive approach of TS1 obviates the need for this entirely because each cache line is *explicitly* in the correct state.

TS as proposed enforces coherence on the whole array level. TS1 can be used to enforce coherence at the finest available resolution of compiler analysis. This is no worse than the whole array level and often better.

Another way to view this distinction of different strategies is the manner in which global information is passed. In FSI and LSS, global knowledge is never passed. In TS, global knowledge is passed implicitly by each processor incrementing an array clock for those arrays which might have been modified. In TS1 and PEI, global knowledge is passed implicitly by invalidating a section of memory that could have been written on a different processor. For local strategies, there is no way to avoid the invalidate because it is responsible for conveying the global information. The invalidate can be implicit, explicit, pessimistic, or reasonably precise, but it still has the same function. We summarize costs and capabilities of the different strategies after discussing TS' (table 9.3).

7.2.5 Example

Figure 7.1 shows a program annotated with the output from a TS1 compiler and the run-time cache behavior of that resulting program (assuming processor 1 gets iteration1). `DOALL`s are expanded into the work-sharing part (`PDO`) where each processor gets some number of iterations, the common part that all processors execute, and the `BARRIER`,

```
PDO I=1,N
  DO J=1,N
    B(I,J)=A(I)+1
  ENDDO
ENDDO
INV (B(1,1),B(N,N))

BARRIER

PDO I=1,N
  B(I,1)=0
ENDDO
INV (B(1,1),B(N,1))

BARRIER

PDO I=1,N
  DO J=1,N
    C(I,J)=B(I,J)+A(I)
  ENDDO
ENDDO
INV (C(1,1),C(N,N))
```

Processor 1					
A(1)		B(1,1)		B(1,2)	
Valid	Epoch	Valid	Epoch	Valid	Epoch
0	0	0	0	0	0
1	1	1	1	1	1
1	1	1	1	1	1
1	0	1	0	1	0
1	0	1	1	1	0
1	0	1	1	1	0
1	0	1	0	1	0
hit		hit		hit	

Figure 7.1: TS1 Example

which is the end of the DOALL. Applying the TS1 compile-time phase inserts the INV's (invalidate instructions). For each of the three epochs, there is an INV to cover what was written in that epoch.

At the end of the e_1 PDO, A(1) and B(1,*) have been referenced on p_1 in this epoch. Therefore, all cache lines holding these values are valid and have the *epoch* bit set. The INV then removes everything written on p_2 from p_1's cache. If B(2,1) were present on p_1 it would be removed at this point. B(1,1) however has its *epoch* bit set and stays in the cache on p_1. All *epoch* bits are reset at the barrier. After the e_2 PDO, only B(1,1) has its *epoch* bit set on p_1 since it was the only reference. The invalidate does not reference A or columns of B other than the first. Therefore, all of those stay in cache. The invalidate of B(1,1) finds the *epoch* bit set and leaves it in cache. In e_3, all references are then hits.

Using the same example for TS, the write to `B` in the second epoch would cause all columns of `B` to be invalidated. Thus, in the third epoch, all but one element of `B` would be a miss on p_1.

For LSS, `A` would suffer the same as the other columns of `B` and would miss in the third epoch.

For PEI, the second epoch would be handled perfectly by only invalidating the first column of `B`. However, the first epoch would leave only the last column in cache. In the third epoch, the first and last column of `B` plus all of `A` would hit. The rest of `B` would miss.

7.3 Performance

The execution time of TS versus TS1 depends on two factors, the hit rate and the additional cycles used by a more sophisticated invalidate. We present experimental data on the former. We analyze the latter using reference traces from a real program combined with a hypothetical implementation of an invalidate.

We compared TS and TS1 on a small test suite of scientific Fortran programs [Appendix B]. These were chosen because they were available, familiar to the authors, and easily convertible to use with simulator. Our methodology was to apply the TS and TS1 algorithms by hand to parallel Fortran programs. For TS1, the same invalidate calls were added at the end of each epoch as the compiler would have produced. We assumed the compiler could recognize only affine subscript expressions. For TS, invalidate calls were applied to whole arrays in an epoch for which the array appeared on the left hand side of an assignment. This has the same effect on hit rate as the suggested TS implementation. These modified programs were then run through the RPPT [18] simulator. This simulator operates by modifying the assembly code to trap at every global memory reference which is then passed off to a particular architecture simulator.

For identical runs of the test programs, we compared TS, TS1, and hardware coherence. For hardware coherence, we simulated write back caches with an invalidate protocol (WB).

Cyclic work distributions were used. Statistics reflect only shared data and not local data or instruction caching. One word cache lines and infinite caches were simulated so that no evictions occurred due to cache size or organization limitations.

	Procs	Size	Shared Data TS	TS1	WB
LU	10	100	88.7	89.8	90.8
Heat Flow	20	60	62.6	63.5	63.5
Direct[2]	4	4	97.1	97.1	97.7
Erlebacher	10	20	68.4	82.0	82.0

Table 7.2: Hit Ratios (%) for different Strategies

7.3.1 Hit rates

Table 7.2 shows the hit rates for our test suite. The shared data hit rates are for that part of the data that coherence is concerned with. For comparison, the total program hit rate (including private data) is shown for two of the test cases where it was high. Similar relative hit ratios resulted from different combinations of processor and block sizes, except for extreme cases. For larger problem sizes with evictions, the gap narrows.

In both LU and Heat Flow, TS1 managed to find extra hits by not invalidating the whole array in loops that set border elements of the (sub-)arrays. Direct made heavy use of indirection arrays which defeat all attempts at analysis. TS1 could do no better than TS in this case. For Erlebacher, TS1 was able to find substantial benefit because the main computation was distributed through several loops, many of which only modified a small section of a given array.

7.3.2 Invalidate overhead

We focus on Erlebacher since it is more than a computational kernel (Direct is also but its use of indirection arrays forced TS1 to treat it the same way TS would).

To better analyze this case, we looked more carefully at the *miss margin*, the number of extra misses per processor that TS suffers compared to TS1. For a fixed problem size, TS1 does worse as the number of processors increase because the total hit rate is only slightly affected causing the number of misses per processor to drop almost linearly. For a fixed number of processors, TS1 is favored by the same reasoning (for all but the simplest of invalidate implementations). Most importantly, as main memory latency increases TS1 is favored. For an invalidate cost model (§7.2.2), we assumed that a contiguous section can be invalidated in $1+\left\lceil \log_2\left(|section|\right)\right\rceil$ "invalidate cycles" (by "invalidate cycle" we

[2]Hit rate figures for Direct include private data.

Procs	Size	Refs	TS hits	TS1 hits	Miss Margin	Cost	Penalty
		1,000's / processor					
1	2	0.44	0.26	0.34	0.09	0.35	0.32
2	4	1.33	0.85	1.11	0.25	0.69	0.35
3	6	2.82	1.82	2.28	0.45	1.36	0.38
4	8	4.84	3.21	3.97	0.75	2.33	0.38
5	10	7.38	4.96	6.06	1.11	3.21	0.38
6	12	10.46	7.06	8.58	1.52	4.81	0.41
7	14	14.06	9.54	11.54	2.00	6.15	0.42
8	16	18.19	12.38	14.92	2.54	7.75	0.42
9	18	22.84	15.59	18.74	3.14	9.44	0.42
10	20	28.03	19.17	22.99	3.81	12.74	0.42
11	22	33.74	23.12	27.67	4.55	15.04	0.42
12	24	39.98	27.44	32.78	5.34	17.52	0.42
13	26	46.75	31.96	37.71	5.75	20.20	0.44
14	28	54.04	39.96	43.83	6.87	23.08	0.45
15	30	61.87	41.28	50.00	7.82	26.21	0.45

Cost: Invalidate cycles for $\log_2(s_i)$ metric

Penalty: Invalidate cycles for whole array invalidates

Table 7.3: Erlebacher Profitability

mean the time it takes to invalidate a single, aligned, power-of-2-sized block) . For sections like `A(1:N,1)` this takes O(log `N`) time, but for sections like `A(1,1:N)` (for a column major language) it takes O(`N`) time.

We applied this cost model to a series of Erlebacher runs where the problem size was varied from 2 to 30 as the number of processors varied from 1 to 15. We chose this as a natural scalability condition because the hit rate for TS stayed fairly constant with this condition. The raw data is summarized in table 7.3. For each run, it lists the total shared data references, the miss margin, the invalidate cost (for invalidating precise sections), and worst case TS1 penalty (in invalidate cycles). All of these are normalized to be per processor. Profitability depends on the relative cost of a miss versus the cost of an invalidate cycle. Figure 7.2 shows the profitability region for some hypothetical miss costs. The profitability is expressed as the percent speed up across all accesses to shared memory addresses (private accesses and non-invalidate instruction costs are the same for TS and TS1).

For this test case, if the cost of a miss is one invalidate cycle (i.e. 0 penalty since cache hits cost one cache cycle) then careful invalidation gains nothing and loses in overhead. If the cost of a miss is as little as 10 cycles TS1 improves performance by paying for the invalidate overhead with time saved from more cache hits.

For miss costs so low that TS1 is slower than TS, it is possible to switch from a precise invalidate to one which invalidates the whole array. This has the same hit rate as TS, but greatly reduces overhead. For instance, in the 15 processor case, the overhead of invalidating whole arrays is about 0.7% (of shared-memory access time). This overhead is always paid, even if every reference were a hit. The "whole array invalidate" line in figure 7.2 represents this. In no circumstance will TS1 perform worse than this with respect to TS.

7.4 Other Issues

7.4.1 DOACROSS

The assumed model for `DOACROSS` is that later iterations (instances) can wait on earlier iterations. There can be multiple waits on the same previous iteration or several different previous iterations. The compiler cannot necessarily determine anything about the nature of the synchronization. The only guarantee is that no dependences go from later to earlier iterations. A legal schedule for any `DOACROSS` would be to do the first p iterations with proper posting and waiting, then synchronize all p processors, and do the next p iterations, etc. Unlike `DOALL`, there can be carried dependence between iterations of a `DOACROSS`. Therefore, each instance of a `DOACROSS` epoch must conceptually be treated like a different mini-epoch.

The leverage that local strategies get from the semantics of `DOALL` (§3.1.2) must be abandoned here. For `DOACROSS`, the *epoch* bit in TS1 indicates that values can be reused on subsequent instances (not epochs). Since two subsequent iterations are almost certain to be scheduled to different processors, the *epoch* bit is useless.

Of the previous strategies surveyed, only Min and Baer's TS [50] handles `DOACROSS`. It increments the version number for an array at the end of a `DOACROSS` epoch. To preserve the semantics inside of the `DOACROSS`, any reference which could be overwritten in a later instance of the same `DOACROSS` epoch, is marked so that it will be removed at the end of the instance. Conversely, any read which could have been preceded by a write in a previous instance is invalidated on entry to this instance (intra-instance locality is still preserved).

Min and Baer's TS still preserves inter-epoch reuse if it can be proven (via the best available compiler analysis) that a given access will not be over-written on a later instance in the same epoch. Likewise, a read need not be forced to miss if it can be proven that no write on a previous instance of this epoch could reach it. Min and Baer's TS handles this situation with extra bits in the instruction word to specifically mark this condition. This is no longer optimal, even in the restricted sense of local strategies being optimal. Certain kinds of inter-instance intra-epoch reuse could be recognized by a local strategy, but are lost here.

TS1 could work in essentially the same way as TS for `DOACROSS` by adding the same extra bits. These extra bits could be avoided by changing the invalidation strategy. Instead of invalidating at the end of each epoch, the invalidate could be moved to the start of each instance. The invalidate would then handle those values written since this processor was last scheduled. For instance, if processors are assigned to iterations in strictly cyclic order and there are 5 processors. Then, processor 5, when it gets assigned iteration 11, would invalidate everything written on iterations 7 through 10. Iteration 6 writes were previously performed on processor 5 and do not need to be invalidated. Iterations before 6 were handled when processor 5 was assigned iteration 6. At the end of the `DOACROSS`, every processor must invalidate writes that occurred since it was last scheduled. This preserves the same inter-epoch reuse as Min and Baer's TS strategy.

In some cases involving `DOACROSS`, a *live* value is guaranteed to be invalidated before its next reference. In this case, there is no need to allocate a cache line. Read-through and write-through could selectively be used to advantage. Min and Baer [50] discuss this at length. This can be done with the bits already present in their strategy. TS1 could accommodate this with extra instruction bits performing the same function as in the Min and Baer strategy.

7.4.2 Critical Sections

For critical section semantics that require inter-instance dependences to be entirely within critical sections, it is a simple matter to maintain coherence. Whatever is written in the critical section must be updated before the end of the section. Whatever is read in the critical section that could be written in another critical section must be invalidated on entry to the critical section.

7.4.3 Non-Unit Cache Lines

If compiler stratagems cannot avoid false sharing because of multi-word cache lines, a local strategy must still operate on *logical* one word cache lines. This does not preclude the physical cache line from staying the same. The benefit of spatial locality is retained. It does require the *epoch* and *valid* bits to be duplicated for each word. Tag bits and age bits still only need to be per cache line. With these changes, everything already discussed goes through as before. We consider this issue again in more depth in connection with TS' (§8.8).

$\mathtt{A}_{ac}[\mathrm{i}]$	$\mathtt{A}$ is accessed on p_i
$\mathtt{A}_{ac}$	$\mathtt{A}$ is accessed on some processor
$\mathtt{A}_{ac}[\neg\mathrm{i}]$	$\mathtt{A}$ is accessed on p_j, $i{\neq}j$
$\mathtt{A}_{ac}$	$\mathtt{A}$ appears to be accessed at compile time
$\neg\mathtt{A}_{ac}$	$\mathtt{A}$ is not accessed, known at compile time
$\neg\mathtt{A}_{ac}$	$\mathtt{A}$ is not accessed, but may appear to be so at compile time
Some implications:	
$\neg\mathtt{A}_{ac}[\mathrm{i}]$	$=$ $\neg\mathtt{A}_{ac}$ or $\mathtt{A}_{ac}[\neg\mathrm{i}]$
$\mathtt{A}_{ac}$	$=$ $\mathtt{A}_{ac}[\mathrm{i}]$ or $\mathtt{A}_{ac} \wedge \neg\mathtt{A}_{ac}[\mathrm{i}]$
$\mathtt{A}_{r}$	$=$ $\mathtt{A}_{r}[\mathrm{i}]$ or $\mathtt{A}_{r} \wedge \neg\mathtt{A}_{r}[\mathrm{i}]$
$\mathtt{A}_{w}$	$=$ $\mathtt{A}_{w}[\mathrm{i}]$ or $\mathtt{A}_{r}[\mathrm{i}]$ or $\mathtt{A}_{w} \wedge \neg\mathtt{A}_{ac}[\mathrm{i}]$ (by §3.1.2)

Table 7.4: Reference Types

7.5 A Formalization of Ideal Dynamic Local Coherence

With the implementation of TS1 in mind, this section demonstrates that the definition of ideal dynamic local strategies is the right one and that TS1 is such a strategy. For this section, we extend the notation previously used (table A.1) to make explicit the distinction between compile-time and run-time knowledge (table 7.4).

The definition of run-time staleness (definition 3.2) is then:

- RTS: $\mathtt{A}_{ac}[\mathrm{i}]\ (\mathtt{A}_{ac}[\neg\mathrm{i}])^{*}\ \mathtt{A}_{w}[\neg\mathrm{i}]\ (\mathtt{A}_{r}[\neg\mathrm{i}])^{*}\ \mathtt{A}_{r}[\mathrm{i}]$

where $\mathtt{A}$ is understood to be an array or a section of an array as appropriate to the level of compile-time analysis available. Next, consider the two limitations on compile-time analysis, concisely expressed:

- $A_{ac}[i] \Rightarrow A_{ac}$
- $A_{ac} \Rightarrow A_{ac}$

These are not just a matter of notation. The first says that a real reference which occurs on a given processor at run-time must be assumed to occur on any processor by the compiler. That is a basic assumption of shared-memory multiprocessor environments (§3.1.2). The second says that any actual reference will be an apparent reference to the compiler, but leaves open the possibility that there may be additional apparent references which do not really occur. Plugging these two limitations into the definition of run-time staleness yields:

- $A_{ac}\ (A_{ac})^{*}\ A_{w}\ (A_{r})^{*}\ A_{r} =$
- CTS: $A_{ac}^{+}\ \ A_{w}\ \ A_{r}^{+}$

which is the definition of compile-time staleness (definition 3.3). The next step is to incorporate local run-time schedule information into this model. Given two references to the same variable in two different epochs, local information can determine that they were referenced on the same processor, if both references actually occurred. Local information alone *cannot* determine that two references occurred on different processors (because the other one might not have occurred at all) . Stated symbolically, the truth of the following (for a given word) can be determined by local schedule information:

- LS:
- e_f:$A_{ac}[i]$, e_g:$A_{ac}[i]$
- e_f:$A_{ac}[i]$, $\neg e_g$:$A_{ac}[i]$
- ~~e_f:$A_{ac}[i]$, e_g:$A_{ac}[\neg i]$~~ – this cannot be determined by LS

To avoid clutter with epoch number references, we will express this simply as $A_{ac}[i]$ $A_{ac}[i]$ and $A_{ac}[i]$ $\neg A_{ac}[i]$ with the understanding that the *n*th term in each expression refers to the same epoch. '*' is a placeholder for a *single* epoch.

If LS is true for the essential access, A_{ac}, and essential write, A_{w}, of an AT, then RTS cannot be true. Similar comments apply for A_{w} versus A_{r}. If either A_{ac} or A_{r} does not actually occur on p_i there is nothing to invalidate on p_i since RTS cannot be satisfied. An ideal local coherence strategy can recognize as stale those things determined stale at compile time minus those things specifically proven not stale by LS. Focusing on a given processor, p_i, an apparent access, A_{ac}, either definitely occurs on that processor, i.e. $A_{ac}[i]$, or it does not but still appears to occur somewhere, i.e. $\neg \mathtt{A}_{ac}[i] \wedge \mathtt{A}_{ac}$. This observation lets CTS and LS be combined to formalize ideal local staleness:

ILS:

CTS - LS = [ignoring $(\mathtt{A}_{ac})^*$ & $(\mathtt{A}_r)^*$ as unaffected terms]

$$
\begin{aligned}
&\{\mathtt{A}_{ac}\ \ \mathtt{A}_w\ \ \mathtt{A}_r\} \\
&\quad - \{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ *\ \} - \{\ *\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r[i]\ \} \\
&\quad - \{\ *\ \ *\ \ \neg\mathtt{A}_r[i]\ \} - \{\ \neg\mathtt{A}_{ac}[i]\ \ *\ *\ \} = \\
&\{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r[i]\ \} + \{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \mathtt{A}_w[i]\ \ \mathtt{A}_r[i]\ \} + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad - \{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ *\ \} - \{\ *\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r[i]\ \} \\
&\quad - \{\ *\ \ *\ \ \neg\mathtt{A}_r[i]\ \} - \{\ \neg\mathtt{A}_{ac}[i]\ \ *\ *\ \} = \\
&\{\ \mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r[i]\ \} + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad + \{\ \mathtt{A}_{ac}\wedge\neg\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w[i]\ \ \mathtt{A}_r\wedge\neg\mathtt{A}_r[i]\ \} \\
&\quad - \{\ *\ \ *\ \ \neg\mathtt{A}_r[i]\ \} - \{\ \neg\mathtt{A}_{ac}[i]\ \ *\ *\ \} = \\
&\mathtt{A}_{ac}[i]\ \ \mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]\ \ \mathtt{A}_r[i]
\end{aligned}
$$

This is exactly ideal local coherence as already defined (definition 3.7), but stated more formally here. The implementation of this observation is simply:

Theorem 7.1: Ideal Dynamic Local Strategy:

Invalidate $\mathtt{A}_w\wedge\neg\mathtt{A}_w[i]\wedge\neg\mathtt{A}_r[i]$ at the end of each epoch.

This works without any reference to the rest of the AT. Clearly, it captures everything in ILS. Nor does it capture too much. If there is no $\mathtt{A}_r[i]$, then the extra invalidate does no harm since there was no reuse to lose. If there is no $\mathtt{A}_{ac}[i]$, then there will not actually be anything in the cache to invalidate and no reuse will be lost. This is a strong conclusion. It says that the only run-time information which needs to be considered in order to build in ideal dynamic strategy is what was referenced in the current epoch.

The epoch bit in TS1 records $A_w[i] \vee A_r[i]$. The rest of the construction of TS1 exactly implements theorem 7.1 and thus is an ideal dynamic local strategy. TS also implements theorem 7.1 though in somewhat less obvious fashion. If A_w, the clock for A is incremented at the end of the epoch. If $A_w[i] \vee A_r[i]$, the time stamp is updated to be the clock+1 when the reference occurs. The invalidate is implicit in time-stamp < clock so the reference overrides the invalidate implicit in the clock increment.

TS1 can utilize resolution that TS cannot, $|A_w|_{TS1} \leq |A_w|_{TS}$ leaving fewer words subject to invalidation. Since both are ideal at the given resolution, TS1 strictly improves on TS's hit rate.

As discussed previously (§3.1.3), there is one piece of the schedule which might be known at compile-time: that serial code executes on a special processor, p_0. This can be incorporated into the model used in this section. Nothing changes except that a few special cases are thrown out.

The analysis in this section is built on the assumption that compile-time and run-time information are collected independently and then combined for invalidation. As previously noted (§3.2.4), this may be too strong of a condition. The compiler can insert code that represents its inferences but leave the final conclusion to be computed at run time. For instance, the compiler might recognize that $B_{ac}[i] \Rightarrow \neg A_w[i]$ then leave it to the run-time system to not invalidate A, if B were actually accessed. We have not seen any cases in our test suite where this would be useful nor is it clear how a compiler might usefully generalize and implement this.

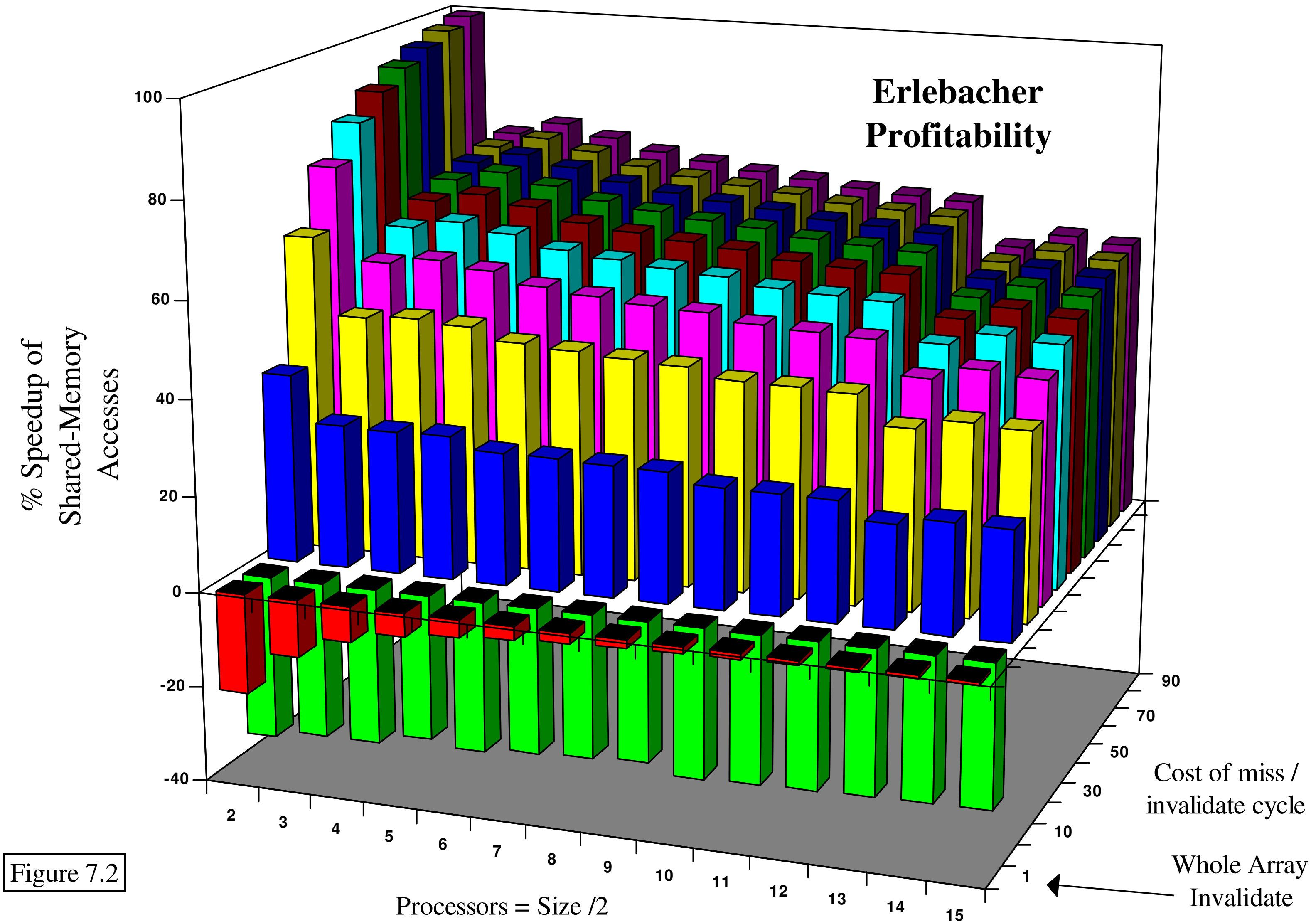

Erlebacher
Profitability
% Speedup of
Shared-Memory
Accesses
100
80
60
40
20
0
-20
-40
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Processors = Size /2
90
70
50
30
10
1
Cost of miss /
invalidate cycle
Whole Array
Invalidate

Figure 7.2

7.6 Fast Invalidate Implementations

A fast invalidate is required to make TS1 effective. This chapter considers some of the alternatives. We address this topic separately without any final conclusion because the implementation of the invalidate is orthogonal to TS1 *per se*.

There are various ways to construct sufficiently fast invalidates. They involve trade-offs between execution time, cache logic complexity, and additional storage. All apparent solutions are expensive in one way or another.

Even a O(log n) invalidate has scalability problems because n is the size for all of the shared data and not just the data local to processor. The effect of this depends on what assumptions one makes about how problem size and number of processors grow together. We examine the impact of different assumptions here.

The fastest direct approach is to have the invalidate simultaneously apply to each cache line (or finite sized block thereof). This can be an exact match using a fully ordered test (i.e. '<', '=', or '>') which works in O(1) time or a mask match which works in O(log n) time.

7.6.1 Exact Match

The fully ordered test would need a fully associative cache where each comparator returns '<' as well as '='. Invalidation proceeds by first broadcasting the low address of a range to be invalidated. Every cache line would record in a special bit whether or not its tag was greater than the low address. Then the high address would be broadcast. If the special bit were set and the tag was less than the high address, the line would be invalidated. The running time would be two cache cycles per range, i.e. O(1) with low constants.

This is clearly impractical though because of the overhead of a tag comparator for every line. Any real world cache (except for the smallest and most critical, e.g. TLB lookup) uses limited associativity, one to four is typical . Also, comparators capable of returning '<' are more expensive than those simply returning '='.

The asymptotic speed could be preserved at the cost of high constants by using a fixed number of cache lines (a 'block') per comparator. For example, it might be reasonable to build a cache with a comparator for each 128 cache lines regardless of the size of the cache. The cost of invalidation approach would then be 256 cache cycles per range. How low this needs to be in order to be practical is yet to be determined. It also imposes constraints on the cache addressing hardware so that all of the addressing to different blocks can happen in parallel.

7.6.2 Mask Match

It is cheaper to build an '=' test than a '<' test in hardware. By using a mask for which bits to respect, this can be used to select a range of addresses. The logic is:

T – number of tag bits
t_i – bit i in the tag
m_i – bit i in the mask (1 means this part of address must match)
a_i – bit i in the address to match
$match_i$ = not((t_i xor a_i) and m_i)
$$match = \sum_{i=1..T} match_i$$

For instance, an address of 0xE0 and a mask of 0xF0 would match all tags between 0xE0 and 0xEF. Lines which match are invalidated. This allows a contiguous address range to be invalidated in O(log |section|)invalidate cycles by knocking out successive power-of-2 blocks. For instance, to clear a range between 0xD8 and 0xF2 would require the following set of invalidates:

Address	Mask	Range Cleared
0xD8	0xF8	0xD8-0xDF
0xE0	0xF0	0xE0-0xEF
0xF0	0xFC	0xF0-0xF1
0xF2	0xFF	0xF2

When a bit is zero in the lower range address or in the upper range address +1, no invalidate power-of-2-block is needed. This happens at least half the time (alignments tend to make it happen slightly more often than a pure random distribution). There are two ends to be invalidated. These two effects cancel leaving $1+\lceil \log_2(|section|) \rceil$ invalidate cycles necessary on average. For those bits which select the cache line are not in the tag, this address and mask logic must also be present in some form.

This is essentially what PEI [47] does. However, it insists that arrays be laid out in such a way that one invalidate cycle is always sufficient. This can waste substantial amounts of memory. Partially arranging for special layouts and strides could greatly reduce the invalidate cost with only minimal impact on memory use. Unlike the exact-match approach, using match logic per block is not acceptable here because the log cost overhead is already so significant that additional constant factors would probably make this approach infeasible.

There exist some commercial content-addressable devices [60] that could be easily modified to provide this functionality. But they are still too expensive for large caches (nor is it likely this will change as we expect standard cache sizes to grow as quickly as the price for these devices drop).

Both the exact-match and mask-match approaches require per cache line logic, and not merely storage. No matter how it is approached this is a significant cost compared to any reasonable amount of per cache logic.

7.6.3 Contiguous Address Ranges

The mask-match approach requires a power-of-2-block of cache addresses to be selected (in addition to a range of tags). It is possible to extend this to simultaneously select any range of addresses. This requires a custom decoder but it needs to exist only for whichever chip contains the valid bit. A normal decoder consists of a tree of individual 2-to-4 decoders (for example, 3-to-8, etc., is also used). Each individual decoder is a 3 input AND-gate for each output line (i.e. four total ANDs for a single block). The extra input is the enable from higher levels. In contrast, a full range comparison decoder requires three enable signals to encode the relevant states, two full comparisons ('<', '=', and '>') and three OR-gates.

Following Abramson [1], this can use a row/column matrix addressing strategy so that each decoder (row and column) need only decode half the address bits. If the decoder is using 2-to-4 decoders (as a trivial example) in a decode tree on a 64K cache, there would be about 350 AND-gates for each the row decode and another 350 for the column decode. At this level, it becomes feasible to perform the full range decode. The penalty is that what took one cycle now takes three cycles. A contiguous range instead becomes (in column major order):

- A range of columns for which all rows are selected.
- A single column for which the leading rows of the section are selected.
- A single column for which the trailing rows of the section are selected.

This three way split repeats itself on another level, tag match versus cache address. For example, if the address space is 32 bits, a cache line is 32 bytes (5 bits of address), and the cache is 64K direct mapped, then the tag would be 16 bits. For large invalidates that may have more than one tag involved, every cache line is selected while the log iteration process matches tags. When < 64K of data (on either end of the original section) remains to be invalidated, there will only be a single tag to match, but the range decode will become selective.

7.6.4 Full Maps

Another possibility is to construct a full map of all cache line addresses in shared memory space, one bit for each cache line. The range decode mechanism could then be used to directly invalidate a region in 3 cycles (at most). This is fast and has a simple implementation (completely avoiding per line cache logic). However, the memory requirements are substantial and not linearly scalable. Such high overhead costs have been contemplated for early global directory strategies though [3]. Such an approach would clearly ruin one of the benefits of local strategies, perfectly scalable memory requirements. However, it would still retain the advantage of no coherence contention delays and not interfere with scalability in that way.

7.6.5 Background Invalidates

Multi-ported caches are used in some designs, either to accept invalidation requests from the memory or to handle prefetch requests. Prefetching often assumes that multiple instruction slots are available. In such a situation, it would be possible to issue invalidates (of any sort, including single lines) as well as prefetches. These invalidates could then proceed in parallel with the computation.

The invalidates would not actually take effect, but only set a provisional bit, PB. If they were to take effect it would be possible that a value would be invalidated before it was actually referenced thus inducing an extra miss. At the end of an epoch, the valid bit gets the epoch bit only if PB. This preserves exactly the same amount of reuse as in the naive strategy. If the PB were to be set for a line not in cache but yet to be referenced, no harm is done, because its epoch bit will be set at the end of the epoch anyway. That its PB bit was never set doesn't matter.

This has two drawbacks. There is no obvious way to interleave invalidate operations with normal references. The computation of the relevant addresses is *not* redundant nor are the addresses to be invalidated fixed constants. Thus some register interference is necessary and it could easily be greater than for the invalidate at the end approach Second, the number of lines to invalidate can actually be larger than the number of words referenced (in a TS1 strategy).

If the epoch following an essential write of an AT does not reference the section, the invalidates can be moved there. That ameliorates the above two problems. A special case is when the next epoch is a serial epoch. The invalidates for non-root processors can be done in what would otherwise be idle cycles. The invalidation pattern for the root processor will often be smaller and not become a bottleneck.

Increasing the associativity or cache line size decreases the amount of time that this takes. Implementing this background invalidate as a simple line-at-a-time invalidate could also be useful in conjunction with other kinds of invalidates. Small sections not following some simple layout incur high overheads for the mask type invalidate since the log factor can not be brought to bear, but they can easily be handled in the background.

7.6.6 Validates instead of Invalidates

Another possibility is to have a validate instruction instead of an invalidate instruction. At epoch boundaries, everything with the epoch set would stay valid as before. However, everything else would be provisionally marked invalid. In the next epoch, a background task, like in the previous section, above would start *validating* cache lines.

It would validate everything that appears to be referenced in the current epoch so long as it does not validate anything which was written in the previous epoch (and would normally be subject to invalidation). The value in this approach is that the validation process does not have to finish. It simply does as much as it can. If there is a line which is referenced, and should be valid, but has not yet been validated, it simply misses and correct execution is still assured. Similarly, the estimate of what is referenced in this epoch can be approximate in either direction (too little or too much).

The number of sections for validation is the same as the number of sections for invalidation (+1). But, they are probably larger in memory address space. They are not obviously larger in terms of actual cache lines used (especially if data is laid out with this in mind with shared writable data grouped together).

7.6.7 Asymptotic Costs of Invalidate

Let a given program be characterized by:

- s – size of run in total data elements
- p – number of processors
- mm(s,p) – miss margin
- $INV_A(s,p)$ – invalidate cost based on log(|array|) metric
- $INV_V(s,p)$ – invalidate cost based on log(|vector section|) metric
- $INV_C(s,p)$ – invalidate cost based on log(|contiguous|) metric

For a fixed s, O(mm(s,p))=O(1/p), i.e. the number of processors does not change the *total* miss margin. For a fixed p, O(1) < O(mm(s,p)) <= O(s), i.e. the miss margin grows with the problem size but it does not grow as quickly as the total problem size. Both of these are empirical observations and need not always be true. Still, they confirm intuition. Whatever benefit TS1 provides over TS is based on the layout of the data, not on how many processors are applied to the layout. TS1 preserves some sections that TS does not. As the problem size increases, those sections represent at most the same fraction of references. But, often some of them will represent asymptotically less, e.g. a column is saved in a matrix and the improvement grows only as the square root.

For a fixed s, O(INV(s,p))=O(1) since a given invalidate must invalidate everything written in a loop regardless of the number of processors used, except for minor overhead costs. Where these are not relevant, we refer simply to INV(s). Because of this TS1 does worse as the processors are increased because the miss margin per processor falls while the total invalidate cost stays the same.

<u>INV_A Metric</u>

As s increases the miss margin increases simultaneously with the invalidate cost. Whether increasing size favors TS1 depends on whether or not O(INV(s))<O(mm(s,p)). This is the case for all real programs except those for which mm(s,p)=0. This does not say TS1 is an

improvement in every case, only that it is more likely to become so as the problem size increases for a given number of processors.

The more interesting and realistic case is some combination of scaling problem size and processors simultaneously. If s/p is held constant, TS1 improves for $O(s \log(s)) < O(mm(s))$. Super linear increases in the miss margin do not occur for real programs. Therefore, for this scaling assumption, TS1 will look worse for large problem sizes.

s is not an array size (of a single dimension) but total data size. Scaling with s/p constant is not necessarily reasonable. For instance, a matrix multiply might keep the number of rows per processors constant. In that case, s^2/p is constant. Now, TS1 improves with size for $O(s^{1/2} \log(s)) < O(mm(s))$. This is neither generally true nor false.

In general, consider scaling a problem such that s^k/p is constant. Increasing k favors TS1. k=1 is a sure loser for a $O(\log n)$ invalidate. $k=\infty$ is almost surely a winner. Typical values of k=2 and k=3 are profitable for some programs and not for others. There is some reasonable expectation of scalability under these assumptions. Unfortunately, there is no obvious way to implement the INV_A metric while preserving the full miss margin.

<u>INV_C Metric</u>

The INV_C metric can be implemented as already discussed (§7.6). In this case, the actual cost of TS1 depends on what fraction of s increases section size and what fraction increases the number of sections. The exact amount will be even fuzzier because different loops and different arrays will affect it differently. Still, a good approximation is to assume there is some intrinsic dimension of the problem, d. The size of the sections then increases by $s^{1/d}$ and the number of sections by $s^{1-1/d}$. The cost of TS1 is then $O(s^{1-1/d} \log s^{1/d})$. Using k as a scaling factor, a $O(\log n)$ invalidate asymptotically improves for $O(s^{1/k}\, s^{1-1/d} \log s^{1/d}) < O(mm(s))$. Thus $1-1/d+1/k < 1$ is needed. This reduces to $d<k$ (assuming $O(mm(s))$ never has any log factors).

Unfortunately, $d \geq k$ for most real programs. The processors might split the problem in every dimension or less. But no program increases processors faster than total problem size. Thus, under these assumptions INV_C will be asymptotically unprofitable.

This is not always the case though. Some programs may increase the section size by $s^{2/d}$ and the number of sections by $s^{1-2/d}$. In this case, asymptotic profitability reduces to $d<2k$. For instance, if a 3-D array were being scanned in such a way that both of the first two dimensions (in column major order) were scanned in every loop and processors scale with the number of rows, these assumptions would be met with d=3 and k=3.

The INV_V metric splits the difference between the INV_C and INV_A metrics. For some strides it can be implemented in a manner similar to that for INV_C. We have not explored this in any detail.

Chapter 8

TS' – An Approximate Invalidate Using Declared Regions

Any invalidate which requires more than O(1) time will be useful only in restricted circumstances (§7.6.7). A general O(1) section-based invalidate is unlikely to have a simple implementation (§7.6). The reason is that O(1) time requires a parallel search on a tag representing the region.

However, there is one type of parallel marking that can be done efficiently in O(1) time, clearing all of the bits on a given chip (§8.7.1). This can be turned into an invalidation strategy. A bit column represents a given region. An invalidate clears the appropriate bit column and by implication all cache entries in that region. It then becomes a compile time problem to assign regions to references so that coherence is achieved, as much reuse as possible is preserved, and the number of regions is kept small. Sufficient regions leave this an ideal local strategy. Insufficient regions produce correct, but non-ideal, programs. They can still be as ideal as possible, in the sense that the total cost of lost reuse is as small as possible for the given number of regions. TS' is an implementation of this approach.

Section 8.1 gives a simplified example of how TS' works and what a region is. Section 8.2.1 discusses how TS1 can be viewed as the same approach, but with unlimited available regions. Section 8.2.2 discusses how TS is lazy and that by making it aggressive, the region name space can be collapsed leading to TS'.

Section 8.3 discusses the core of the analysis which is needed for TS'. The central construction is to convert ATs over the *epoch graph* (program nodes) into a *coloring graph*. Minimizing the cost of coloring this graph then corresponds to assigning regions and placing invalidates. A naïve construction leads to $O(n^3)$ size coloring graphs (for each array section). Section 8.3.1 formalizes the *coloring graph* and proves several theorems that identify *useless* ATs which can be removed from the epoch graph during construction without affecting the final coloring. This leaves the graph O(n) in size and construction time. It also proves that a consistent coloring exists.

Section 8.3.2 proves some more reduction theorems about the *coloring graph* itself (as opposed to the original epoch graph). The net result is that the number of essential components involved in the final coloring is linear in the number of the epochs that read a given variable, summed over all variables.

Section 8.3.3 discusses how to handle inter-epoch loops (i.e. serial loops around parallel loops). Loops with constant reference patterns can have optimal cyclic colorings computed simply by unrolling five times. Handling loops with non-constant reference patterns is also discussed.

Section 8.3.4 addresses the question of write reuse. In most parts of this analysis, it is assumed that writes and reads both cause reuse because the other words in the cache line will be preserved, i.e. write misses stall for a main memory access just as read misses do. If there is no write reuse, the same mechanism as so far presented should be used, with the deletion of the appropriate RAs in the *coloring graph*. Handling writes without reuse by a special mechanism would be more expensive than using the general TS' one.

Section 8.3.5 discusses how the root processor is unique. It's schedule is partially known because it executes serial sections. This information can be used in some cases to improve the quality of the coloring.

Section 8.4 shows that solving *coloring graphs* which arise from simple (sequential) fork-join inter-epoch control flow (i.e. loop-free structured code, not general DAGs) is NP-complete. The complexity of coloring linear graphs remains an open question.

Given that coloring is ultimately an NP-complete problem, some type of exhaustive algorithm will be necessary for an optimal coloring. Section 8.5 presents a pseudo-data flow framework for building ATs and a branch and bound algorithm that optimally solves the corresponding *coloring graph* problem. An implementation of the algorithm to handle linear code and loops is discussed. Building on the reduction theorems of sections 8.3.1 and 8.3.2, the *coloring graph* topology is sufficiently reduced that this algorithm runs quickly for all programs in our test suite, despite being exponential in principle. Currently, no implementation for the more general DAG case exists.

If the number of available colors is insufficient to preserve all reuse, the branch and bound coloring algorithm can use any weighting function to determine the priority of preserving reuse on given ATs (for instance, data volume along a path or the path length). The algorithm retains its optimality in the sense that an answer with the lowest possible weighting will be found, even though some reuse will be sacrificed. As four colors have been sufficient to preserve reuse in all of our tests, we have no results on the value of different weighting functions.

Section 8.6 discusses the mapping of *regions* as recognized by the compiler. The compiler operates on each recognizable section at each node as a separate entity. At run-time, these must be handled either as explicit *colors* attached to static references or as or hardware recognizable address *ranges*, which are then mapped to actual *colors*.

Section 8.7 discusses the special hardware needed to implement TS'. TS' needs a 1-bit wide chip for each color, slightly modified hardware to recognize a hit, *RLB*s to do the range to color mapping, and logic to initialize these structures. Some of this hardware is relatively complex but all of the processing that happens per reference can be done in parallel with normal cache operations and should not slow cache or processor speed. The actual invalidation occurs only between epochs and requires some small, finite number of chip clears, a trivial time overhead. As a principal design objective, all operations are O(1) and act purely on local cache information.

Finally, section 8.8 shows how to extend TS' to non-unit cache lines. By using two bits per word, the cache line can be made to have a logical length of one, i.e. individual words can be *fresh*, *valid*, or *stale*. The tag and color bits are still per line. Spatial locality is preserved. Hardware eviction and aging logic are unaffected. Any local strategy must pay this cost to maintain full reuse in the presence of false sharing. While significant, this cost is considerably less than insisting on unit cache lines. The tradeoffs involved in one extra bit per word are also discussed.

8.1 Introduction to TS'

If a single bit column is to represent a region, then the TS1 approach must be changed. It will no longer suffice to name the region at the time of the invalidate. It must be named at the time of the access. Since region naming must span from access to invalidate, it is necessary for the analysis to take inter-epoch information into account instead of merely using local epoch information as in TS1.

Coherence is preserved in the following example:

Code	Region
`PDO I=1,2`	
`   A(I)=I`	1
`   B(I)=I+2`	2
`END DO`	
`PDO I=1,2`	
`   A(I)=A(I)+4`	3
`END DO`	
`INV R1`	
`PDO I=1,2`	
`   B(I)=I+6`	4
`END DO`	
`INV R2`	
`PDO I=1,2`	
`   =A(I)`	
`   =B(I)`	
`END DO`	

Both references to `A` in the first epoch are marked in cache as being in region 1, etc. then `INV R1` removes all of those references from cache. This keeps any stale values from the first epoch from reaching the fourth. Any reuse from the first to the second epoch and from the second epoch to the fourth is also preserved. At the end of the second epoch, the cache on processor 1 might look this:

	Logical Cache Line				
Variable	Cache Tag	Cache Contents	R1	R2	R3
A(1)	100	5			✓
B(1)	110	3		✓	

The `INV R1` does not change anything in cache because the write in the second epoch covered up the access in the first epoch (at least on processor 1). All reuse is preserved. The other case is that processor 1 might get a different schedule on the second epoch. In that case, the cache might look like:

Variable	Logical Cache Line				
	Cache Tag	Cache Contents	R1	R2	R3
A(1)	100	1	✓		
A(2)	101	6			✓
B(1)	110	3		✓	

Now `INV R1` will simply clear the check in bit R1 for `A(1)`. If processor 1 references `A(1)` in epoch 4, it will find that no region bit is set and it will take a miss, preserving coherence.

The algorithm is summarized in figure 8.1. Correctness is assured because any essential access of an AT is invalidated before the stale essential read. This requires only that computed ATs be a superset of actual ones. Like TS1, this is also an ideal local strategy.

The advantage of this approach is that the number of regions can be small, four in practice. When more regions are needed than are available in order to preserve all reuse, approximations can be made which preserve correctness.

Code	Hit	Logical Cache Line		
		Cache Tag	Cache Contents	R bits
`INV j`	-	-	-	$\forall_i R_i^j = 0$
reference A_i^j	tag match Ù $\left(\bigvee_{j=1}^{n} R_i^j \right)$	set	set or used	$R_i^k = \begin{cases} 0 & k \neq j \\ 1 & k = j \end{cases}$

j - a given region bit

i - a given cache line

Figure 8.1: Run-time TS' Logic

8.2 Relation to previous strategies

8.2.1 TS1 as TS' with Unlimited Regions

TS1 can be viewed as an extreme TS' implementation. Consider each memory address to be a different region. TS1 then sets the region bit on an access by loading the tag, which encodes the region bit number in binary. A TS1 invalidate then simply clears out a set of region bits in one instruction. Since TS1 can use as many regions as it needs, there is no point in delaying the invalidate to subsume other regions, as there is in TS'.

There is one difference though. Even though both must make static decisions, TS1 can use symbolic analysis based on variables assigned during the RA but TS' cannot because the region must be named at the essential access. Figure 8.2 is an example.

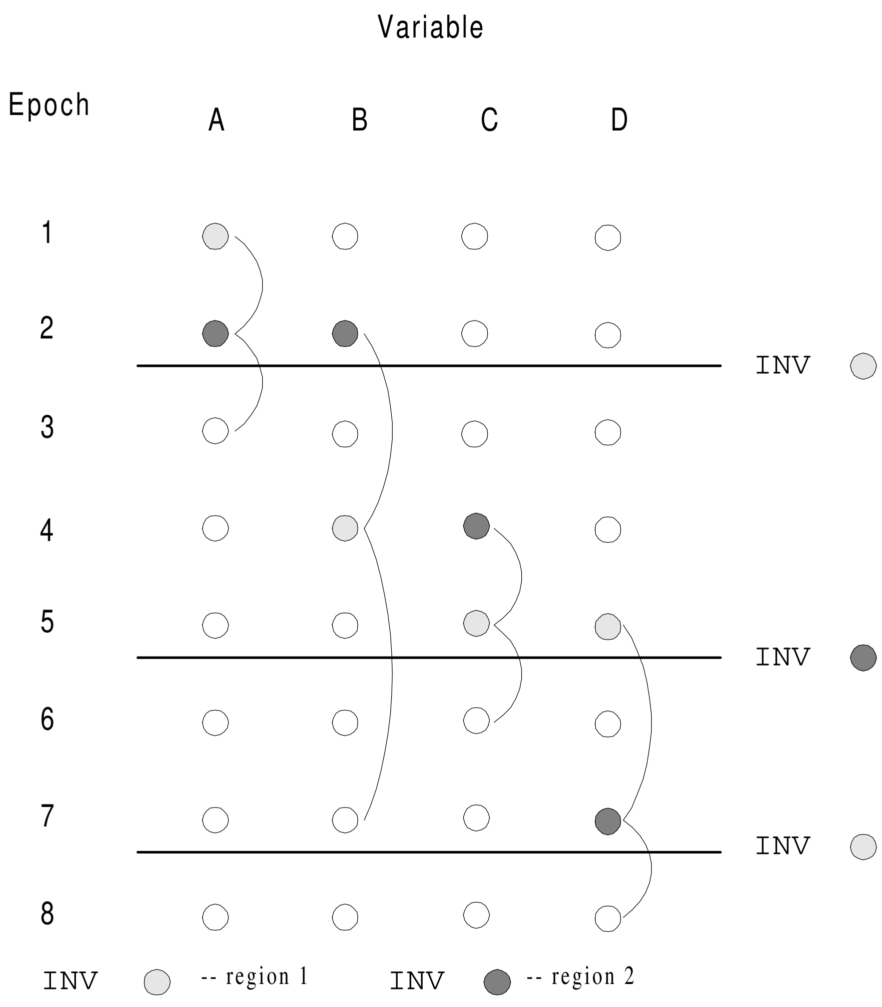
```
PDO I=1,100
   A(I)=      — R1
ENDDO
N=f()
PDO I=1,N
   A(I)=
ENDDO
INV A(1:N)       — TS1
INV 1            — TS'
```

Figure 8.2: TS1 versus TS'

TS1 does strictly better. If `N`>100, then they are identical since the extra elements beyond 100 do not affect reuse. If `N`<100, then INV 1 in TS' also invalidates `A(N+1:100)` unnecessarily. By using addresses as region numbers, TS1 parameterizes for all possible values of `N`.

8.2.2 Laziness as an Essential Feature of Time-Stamping

TS' is more similar to TS than TS1 is to TS. Like TS, TS' numbers the epochs and assigns different run time identifiers for each section of interest. However, TS' is different in an important abstract respect, it actively invalidates stale data instead of letting it be done in a lazy fashion. In TS, no data is removed from cache until the cache line containing it is referenced again (or the whole cache is flushed). There is no upper bound on how long a line might remain in cache. Because of this, it is impossible to recycle region names (until the clocks overflow and the cache is flushed). TS' by virtue of actively invalidating can know with certainty when a region name is no longer in use and can then reuse it. This collapse of the name space allows sections to be handled without an explosion in the number of marking bits, as TS would require.




```
PDO
   A(I)=...
ENDDO
PDO
   A(I)=B(I)
ENDDO
PDO
   ...=A(I)
ENDDO
PDO
   B(I)=C(I)
ENDDO
PDO
   C(I)=D(I)
ENDDO
PDO
   ...=C(I)
ENDDO
PDO
   D(I)=B(I)
ENDDO
PDO
   ...=D(I)
ENDDO
```

Figure 8.3: AT Node/*Epoch Graph* Example of TS'

Even though the distinction can be neatly characterized in the abstract, the implications of it are quite broad. The organization of the hardware, the run-time algorithms, and the compile-time algorithms are all quite different. After discussing TS’ in depth, we return to this issue (§8.7.5) and show how a crippled TS’ with essentially the same hardware as TS is still an improvement.

8.3 Minimizing Regions

The key to making TS' practical is to minimize the number of regions. The cost model and topology developed in conjunction with CTV+ (§6.4) are not adequate here because interactions between different variables do matter. Something considerably more sophisticated is needed.

Minimizing regions starts by converting ATs on the *epoch graph* (e.g. figure 8.3) into a *interference graph* (e.g. figure 8.4) where two nodes are in conflict, i.e. need a different color, if an invalidate at one would destroy reuse at another. The interference graph is augmented to a *coloring graph* (e.g. figure 8.5) which represents some other constraints unique to this problem. A minimal coloring then corresponds to minimizing the number of regions, deciding what color to make each reference, and where to place invalidates.

The epoch graph on the left of figure 8.3 corresponds to the program on the right. Each epoch is duplicated for each variable. ATs are added as arcs at the appropriate places. The color of the nodes represent an assignment of region names. The `INV`s in the right margin of the node graph show where the invalidation for each region takes place. Open circle nodes are no color since it does not matter what color they might actually be. Invalidates are understood to occur immediately after the node they follow. Thus, the `INV` at epoch 2, removes stale values of `A` loaded at epoch1, but does not affect the reuse of $\mathtt{A}_{sa}$ from node 2 to 3 nor the reuse of $\mathtt{B}_{ra}$ from node 2 to 4.

One color will not suffice for this graph. There must be an `INV` on either node 4, 5, or 6 to prevent a stale access on node 7 of whatever part of `B` was loaded on node 2. If this `INV` were on node 4, it would interfere with reuse of `C` between node 4 and 5. If it were on node 5 or 6, it would cause similar problems. The two colors used in the example preserve all reuse making this an optimal coloring.

To create the interference graph, every AT in the epoch graph has three parts in the interference graph, the RA, the SA, and the invalidate. The interference graph has a node for the RA arc in the epoch graph, a node for the SA arc, and a set of nodes of each possible location of the invalidate, referred to collectively as the IA. The RA and SA parts will collectively be referred to as RAs where only the reuse aspect, and not the original AT, matters.

Each possible invalidate node conflicts with the RA node of any variable for which the invalidate would be in the range for the RA (its own RA included). For example, there is conflict between a node 2 invalidate for `A` and $\mathtt{B}_{ra}$. SA reuse is treated similarly (e.g. a node 5 invalidate for `C` would destroy $\mathtt{B}_{sa}$ reuse). The final color of an RA (SA) node represents the color of the v_{ac} (v_w) node for that AT. Coloring then chooses one of each variable's set of invalidate nodes (and all of the RA and SA) nodes. Whichever one is finally chosen is the placement of the invalidate. Since the invalidate must invalidate what was accessed in v_{ac}, the RA node must have the *same* color as whichever invalidate node

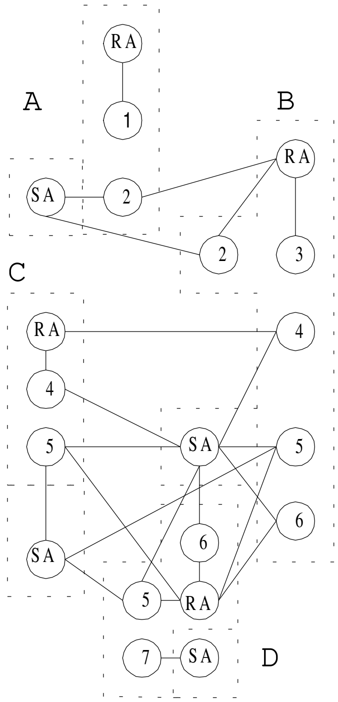


Figure 8.4: Basic *Interference Graph* Example

is chosen. The SA node prefers a *different* color than its associated invalidate to preserve reuse between v_w & v_r. Table 8.1 summarizes these rules plus other definitions needed for the *coloring graph*.

Figure 8.4 shows the basic *interference graph* constructed from these observations. The edge from node 2 in the A box to the RA node in the B box represents the fact that the invalidate for A at epoch 2 would interfere with reuse on B_{ra} if they had the same color. The edge does not interfere with a numbered node in the B box because the invalidate for A has no effect on invalidates for B. The node does not run from A_{ra} because the color of A_{ac} has no direct effect on the color of B_{ac}. The box around B_{ra}, B_2, B_3, B_4, B_5, & B_6 represents, without explicit edges, that one of B_2, B_3, B_4, B_5, or B_6 must be chosen for the INV and then B_{ra} given the same color.

The additional constraints are explicitly added to make a coloring graph from an interference graph. Any RA, SA, or IA which starts at the same node as another (for the same variable) is connected by a "must be same color" constraints, represented by a heavy line. The different possible nodes of an IA are connected by a "must choose one" constraint, represented by a dashed line. This is transitive in the sense that if two nodes are joined by a dashed path (and not just an edge), only one of them should be chosen. Interference is shown, as before, by a thin line.

When only the optimal coloring is of interest (instead of a best possible sub-optimal coloring where weighted decisions matter), the coloring graph can be compressed further. IA nodes which are in the RA can be tossed since choosing to place the INV in the RA must sacrifice some reuse. Those remaining nodes which are joined by only a single edge can be trivially colored and are removed from the coloring graph (e.g. A_{sa} after node 2 of B_{ia} is removed).

Merely as a representational convenience, interference edges from an SA node to its corresponding IA nodes are summarized as a single interference edge to the RA which must have the same color as the IA nodes (e.g. the three edges between B_{sa} and B_4, B_5, B_6, are summarized as a single edge between B_{sa} and B_{ra}). If there is only one IA node for

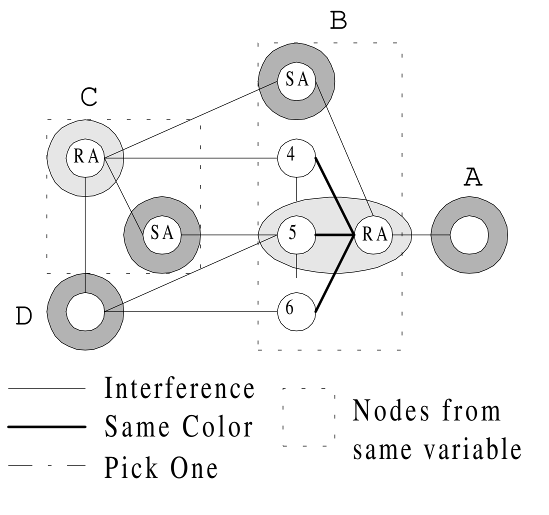


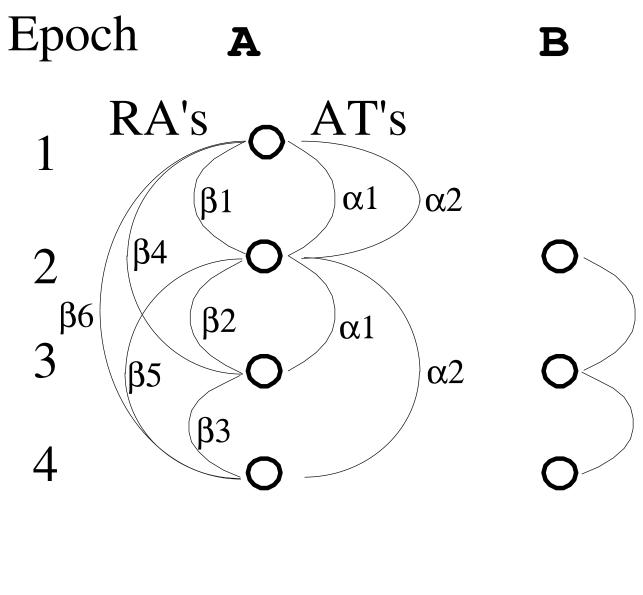
Figure 8.5: Coloring Graph Example

a given RA node, both can be summarized as a single node since they must have the same color and there are no choices to make (e.g. $\mathtt{C}_5$ and $\mathtt{C}_{ra}$ can be lumped into $\mathtt{C}_{ra}$). However, no variable is completely removed in this process. All of the nodes originating from a single variable are enclosed in a box with a dotted boundary. If there is only a single node, the box is omitted. Figure 8.4 with all of these changes is shown as figure 8.5.

Figure 8.5 also shows the final coloring. Nodes 4 & 6 for B are ignored in the final coloring. RA for B is given the same color as node 5. This means that $\mathtt{B}_{ac}$ is in region 2 (the darker region) and the invalidate for B follows node 5. The SA node for B is in region 1 (the lighter region) so $\mathtt{B}_w$ is in region 1 and will not be affected by the invalidate on node 5. The rest follows similarly. Two colors are sufficient to preserve all reuse in this example.

A minimal number of colors preserves full reuse but is not the same as a minimal number of INVs. In this figure, node 6 could have been chosen instead of node 5. This still would have produced a two coloring. However, the invalidate at node 6 for B would not have preserved coherence for C. It would require another invalidate at node 5, but with the same color as the one at 6.

This problem is not like that of coloring for register allocation because two regions do not interfere with each other in a symmetric way. For instance, the INV at node 2 cuts $\mathtt{A}_{sa}$ and $\mathtt{B}_{ra}$. But, any INV of $\mathtt{B}_{sa}$ would have no effect on A. Also, there is a choice of where to place the INV. The final choice affects the interference graph. For instance, the INV at node 4 does not interfere with D. If it were moved to either node 5 or 6, it would.




```
PDO
   =A(I)
ENDDO
PDO
   A(I)=B(I)
ENDDO
PDO
   B(I)=A(I)
ENDDO
PDO
   =A(I)+B(I)
ENDDO
```

Figure 8.6: Example of Multiple ATs per Variable

8.3.1 Multiple ATs per Variable

If there are more than three references to a given variable, multiple ATs may result. These may overlap in several different ways. Figure 8.6 gives an example of multiple ATs for `A`. All of the RAs are on the left of the nodes. The two ATs, $\alpha1$& $\alpha2$, are on the right. As before, an RA has the same color as its source node. In this more general setting, there may be several RAs with the same source node. All of them must have the same color. In figure 8.6, $\beta1$, $\beta4$, & $\beta6$ all must have the same color.

The distinction of RA & SA versus IA becomes clear here. The reuse to be preserved no longer has any convenient relation to the placement of invalidate constraint. For example, `A` has two ATs, $\alpha1$ & $\alpha2$, but six RAs. Because $\alpha1_{sa}$ runs from epoch 2 to 3, there must be an `INV` following node 2. This `INV` also breaks $\alpha2_{sa}$. Even though $\alpha1_{ra}=\alpha2_{ra}$, $\alpha2$ is not irrelevant. If $\alpha2$ were ignored, there would be no conflict between $\beta5$ and B_{sa}. This might result in A_2 and B_2 getting the same color. The `INV` for B at node 3 would then destroy reuse on $\beta5$. The reuse in $\alpha2_{sa}$ ($\beta5$) must be respected even though it is not useful for `INV` placement.

Epoch = $\{i \mid 1 \le i \le N\}$	a set of epochs
$u, v \in V$	program variables
$v_r \subseteq$ Epochs	nodes that read v
$v_w \subseteq$ Epochs	nodes that write v
$v = v_r \cup v_w$, $v_r \cap v_w$ possibly non-empty	nodes that reference v
$\Phi = \{(\alpha_{ac}, \alpha_w, \alpha_r) \in \text{Epoch} \times \text{Epoch} \times \text{Epoch} \mid \alpha_{ac} < \alpha_w < \alpha_r\}$	All possible access triples (ATs)
$\beta = (\beta_{ac}, \beta_{re}) \in \text{Epoch} \times \text{Epoch} \mid \beta_{ac} < \beta_{re}$	A given reuse arc (RA)
$v_{at} = \{\alpha \in \Phi \mid \alpha_{ac} \in v \wedge \alpha_w \in v_w \wedge \alpha_r \in v_r\}$	All access triples (ATs) for a given variable
$v_{ra} = \{\beta \mid \beta_{ac}, \beta_{re} \in v\}$	All reuse arcs (RAs) for a given variable
$v_{ia} = v_{at}$	All invalidate arcs (IAs) for a given variable
$\lvert\alpha\rvert = \alpha_r - \alpha_{ac} + 1$	Definition of size of an arc
$\overline{\gamma} = \{i \mid \gamma_{ac} \le i < \gamma_r\}$	The set of nodes of the span of an IA
$\overline{\beta} = \{i \mid \beta_{ac} \le i < \beta_{re}\}$	The set of nodes of the span of an RA
$\text{Nodes} = \bigcup_{v \in V} \left(\bigcup_{\gamma \in v_{ia}} \left(\bigcup_{i \in \overline{\gamma}} v_\gamma(i) \right) \cup \bigcup_{\beta \in v_{ra}} v_\beta \right)$	Nodes in coloring graph, e.g. $\mathbb{A}_{(1,2,3)}(1), \mathbb{A}_{(1,2,3)}(2), \mathbb{A}_{(1,2)}, \mathbb{A}_{(2,3)}$ (fig. 8.4)
$\text{Edges} = \{(v_\gamma(i), u_\beta) \in \text{Nodes} \times \text{Nodes} \mid i \in \overline{\beta}\}$	A conflict edge runs from an RA to an IA node in its span
color: Nodes $\to \{1,2,3....\}$	A map from nodes to colors
$\beta, \beta' \in v_{ra}$, $\beta_{ac} = \beta'_{ac} \Rightarrow \text{color}(\beta) = \text{color}(\beta')$	same starting epoch, same color
$\gamma \in v_{ia}$, $\exists i \in \overline{\gamma}$,	Definition of preserving coherence
$\text{color}(v_{(\gamma_{ac}, \gamma_w)}) = \text{color}(v_\gamma(i))$	Invalidate must match its RA
$\forall j \in \overline{\gamma} \mid j \ne i$, $\text{color}(v_\gamma(j)) = 0$	Other invalidate nodes are ignored
$\forall (x,y) \in \text{Edges}$, $\text{color}(x) \ne \text{color}(y)$	criteria for preserving reuse
Note: $\neg\exists (x,y) \in \text{Edges} \mid \text{color}(x) = \text{color}(y) = 0$	Ignored invalidates never interfere with each other

Table 8.1: Definitions for Coloring Graph

Definition of Coloring Graph with Multiple ATs

Table 8.1 gives the definitions necessary for the coloring graph in the presence of multiple ATs.

The definition of the set of access triple says that it is a sorted triple of three epoch nodes. The next statement is also meant to convey that all ATs will be referred to as α and their components as α_{ac}, etc. This encompasses all possible ATs which is not to say that all such triples are actual ATs. Reuse arcs are similar but there is no need to name the set of all possible ones.

IAs are identified by triples like ATs, but they are not the same thing. The AT is understood to encompass reuse arcs and possible invalidate locations. The IA is merely the set of locations where an `INV` could occur. The IA spans everything from γ_{ac} to γ_r. The invalidate is understood to immediately follow the node it is identified with. Therefore, the span goes from γ_{ac} to γ_r-1 representing every place to cut *between* the two extremes.

The definition of *Nodes* follows the previous examples. Every reuse arc (RA or SA) is given a node and named by the epochs it spans. For instance, $\mathtt{A}_{(1,2)}$ and $\mathtt{A}_{(2,4)}$ are the node names for the reuse represented by β1 and β5 (figure 8.6). Every possible location of an invalidate is named by the IA it is in and indexed by the invalidate location in the span. For instance, $\mathtt{A}_{(1,2,4)}(2)$ and $\mathtt{A}_{(1,2,4)}(3)$ are the two IA nodes generated by α2. These will often be abbreviated $\mathtt{A}(2)$ and $\mathtt{A}(3)$ where the meaning is clear. It is possible for two different IAs for the same variable to cover the same nodes, in which case the full notation will be used.

The first color property merely says that any two RAs (of the same variable) starting at the same epoch must have the same color since they both represent the same reference.

The coherence property says that the color of the IA node must be the same as that of the RA that it follows and that there must be at least one such node. Which ever node is actually chosen for `INV`, it must invalidate the same region as was used on the original access. Which of the IA nodes is chosen is not relevant here. The other nodes in the IA are explicitly assigned to the color 0. RA nodes must have some positive number as a color. Therefore, the other nodes of the IA that do not represent the final placement of the invalidate do not affect the coloring. Notice that two IA nodes never directly interfere with each other. It does not produce a contradiction in the graph to force IA nodes to be color 0.

The edges run from every IA node to every RA or SA node that contains that IA node. Again, the IA nodes represent single epochs for a given AT. Therefore, there is no question of overlap. A given IA node is either clearly in the span of a given RA (represented by a single node) or not. The syntax is compressed. The expression $(v_\gamma(i), u_\beta)$ means that the first element is any node that can match the v_γ(i) form, i.e. any IA node. The second element is then any RA node. Edges are bi-directional. $v=u$ is possible.

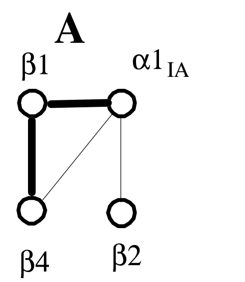


Figure 8.7: Coloring Graph Subset for Figure 8.6.

Resolving Impossible Colorings

With Multiple ATs per variable, graphs which are impossible to color may result. Figure 8.7 shows a subset of the coloring graph for figure 8.6 restricted to the first three nodes (and `A` only). $\alpha1_{ia}$ must have the same color as β1 (=$\alpha1_{ra}$) in order to insure correctness. β1 and β4 must also have the same color since they start at the same node. $\alpha1_{ia}$ and β4 however need different colors because the `INV` at node 2 must surely cut β4. Obviously, no coloring will work. The reason is simply that β4 is impossible to preserve. It is a *useless* RA and can be eliminated from the epoch graph (and therefore the coloring graph).

This theorem, RAT1, can be restated: any RA which *starts* with an AT and completely covers it is useless. It is worth noting that an RA which starts at the same node as an AT but ends in the IA is not necessarily useless. Merely crossing the essential write of an AT does not make an RA useless.

An IA can be useless for the same reason that an RA is, it must be cut by an `INV`. In this case, a *useless* IA has the same first two nodes as another, useful IA, but its third node, the essential access, is later. Also, one IA is crucial for correctness and the others starting at the same node are useless, whereas all cut RAs are useless. In figure 8.6, $\alpha1_{ia}$ renders $\alpha2_{ia}$ useless because it cuts it. This does not render $\alpha1_{ia}$ useless, it is quite essential. Also, this does not render the reuse represented by these arcs useless. β2, β3, & β5 are still useful RAs. They are not necessarily cut by $\alpha1_{ia}$ since node 2 could have a different color than node 1 (which $\alpha1_{ia}$ matches).

$\beta\in v_{ra}$ [β-]useless $\equiv \exists\ \gamma'\in v_{ia} \mid \beta_{ac}=\gamma'_{ac} \wedge$	A *directly* IA-killed RA is useless.
$\beta_{re}\geq\gamma'_{r} \wedge \gamma'$ is useful	The converse is used as a definition.
$\gamma\in v_{ia}$ [γ-]useless $\equiv (\exists\ \gamma'\in v_{ia} \mid \gamma_{ac}=\gamma'_{ac} \wedge$	An IA killed by another IA is useless
$\gamma_{w}=\gamma'_{w} \wedge \gamma_{r}>\gamma'_{r}) \vee ($ (γ_{ac},γ_{w}) is a useless RA)	An IA with a useless RA is useless
	Note that γ' is not necessarily useful
$\alpha \in v_{at}$ useless $\equiv \gamma=\alpha$ is γ-useless	
RAT: $\exists\ \alpha\in v_{at} \mid \beta_{ac}\leq\alpha_{ac} \wedge \beta_{re}\geq\alpha_{r}$	Statement of theorem,
$\Rightarrow \beta\in v_{ra}$ is useless	α is possibly useless
RAT1: $\exists\ \alpha\in v_{at} \mid \beta_{ac}=\alpha_{ac} \wedge \beta_{re}\geq\alpha_{r}$	Start with a weaker theorem
$\Rightarrow \beta\in v_{ra}$ is useless	
IH: RAT1 is true for $\lvert\alpha\rvert\leq$ k	Inductive hypothesis
Base: k=3, α=(i, i+1, i+2)=γ	The smallest possible AT (and IA)
$\sim\exists\ \gamma'\in v_{ia} \mid \lvert\gamma'\rvert < \lvert\gamma\rvert=3$	An IA takes at least two nodes
γ is useful	definition of γ-useless
$\beta_{ac}=\gamma_{ac} \wedge \beta_{re}\geq\gamma_{r}$	hypothesis for RAT1
β is useless	from definition of useless
Induct: for $\lvert\alpha\rvert$=k+1	
Case 1: α is useful	
for $\gamma=\alpha$, $\beta_{ac}=\gamma_{ac} \wedge \beta_{re}\geq\gamma_{r}$	hypothesis for RAT1
β is useless	definition of useless
Case 2: α has a useless RA	
$\beta'=(\alpha_{ac},\alpha_{w})$ is useless	the RA of α
$\exists\gamma'\in v_{ia} \mid \gamma'_{ac}=\beta'_{ac}\wedge\beta'_{re}\geq\gamma'_{r}\wedge\gamma'$ is useful	from definition of useless
$\beta_{re}\geq\alpha_{r} > \alpha_{w} = \beta'_{re}\geq\gamma'_{r}$	RAT1 hyp, def of AT, & def of β'
β is useless	by definition of β-useless on γ'
Case 3: α has a useless IA, γ, and a useful RA	
$\exists\ \gamma'\in v_{ia} \mid \gamma_{ac}=\gamma'_{ac} \wedge \gamma_{w}=\gamma'_{w} \wedge \gamma_{r}>\gamma'_{r}$	from definition of useless IA
$\alpha_{w} = \gamma_{w} = \gamma'_{w} < \gamma'_{r}$	definition of IA
$\alpha'=(\alpha_{ac},\alpha_{w},\gamma'_{r})$ is an AT	α' does not necessarily have a useful RA
$\lvert\alpha'\rvert < \lvert\alpha\rvert$ since $\gamma'_{r} < \gamma_{r}$	
$\alpha'_{ac}=\alpha_{ac}=\beta_{ac}$, $\alpha'_{r}=\gamma'_{r} < \gamma_{r}=\alpha_{r}\leq\beta_{re}$	α' & β fit RAT1 hypothesis
β is useless by IH on α'	recurse on a smaller α
QED for RAT1	

For $\beta_{ac} < \alpha_{ac}$
$\alpha' = (\beta_{ac}, \alpha_w, \alpha_r)$ is an AT
β is useless by RAT1 applied to α'
QED

Theorem 8.1: Proof of RAT theorem

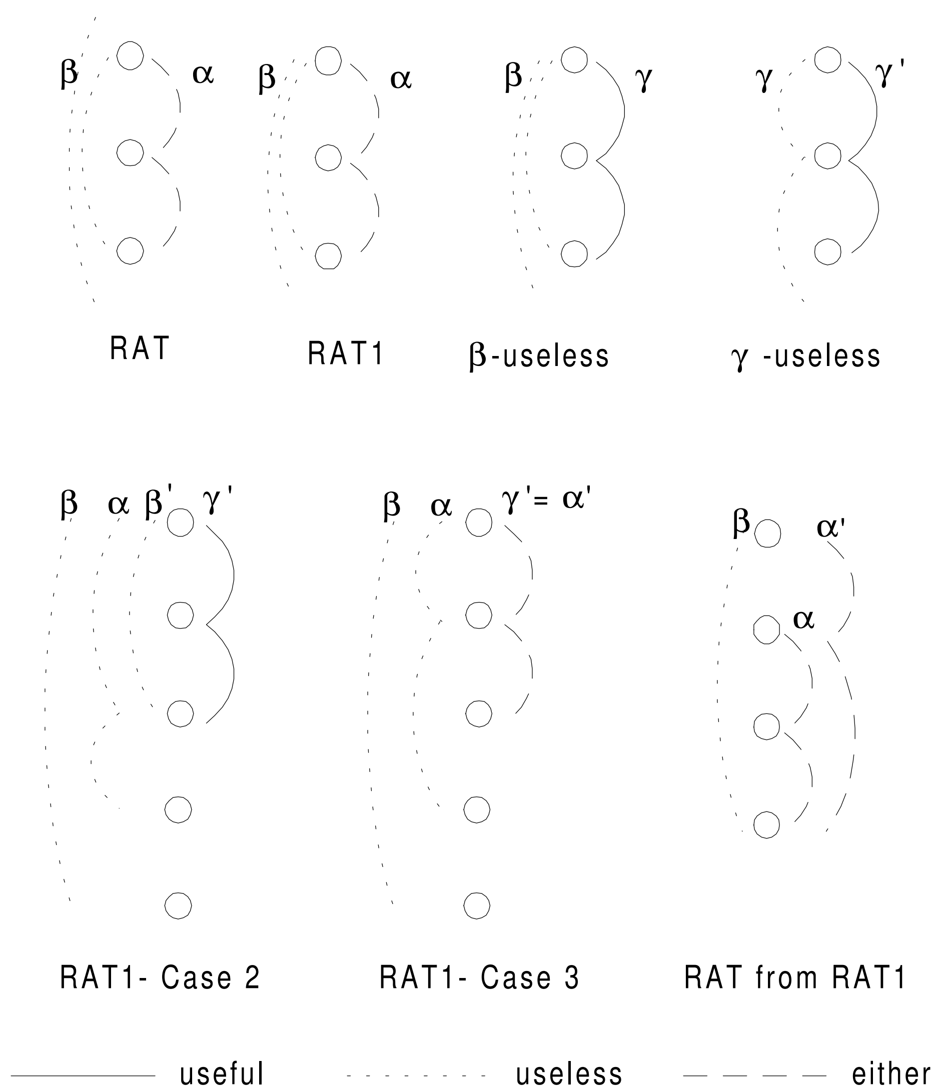


Figure 8.8: Examples for proof of RAT (theorem 8.1)

$(\exists\ \gamma' \in v_{ia} \mid \gamma_{ac}=\gamma'_{ac} \wedge \gamma_w=\gamma'_w \wedge \gamma_r>\gamma'_r)$ $\wedge\ \gamma'$ useful $\equiv \gamma \in v_{ia}$ is IA-useless	Definition of IA-useless
$\gamma \in v_{ia}$ is useless $\equiv$ γ is IA-useless or γ is RA-useless	Definition of useless
IAT: $\exists\ \gamma' \in v_{ia} \mid \gamma_{ac} \leq \gamma'_{ac} \wedge \gamma_r>\gamma'_r$ $\Rightarrow \gamma$ is useless	Statement of IAT theorem γ' is possibly useless
IAT1: $\exists\ \gamma' \in v_{ia} \mid \gamma_{ac}=\gamma'_{ac} \wedge \gamma_w=\gamma'_w \wedge \gamma_r>\gamma'_r$ $\Rightarrow \gamma$ is useless	Statement of IAT1 theorem γ' is possibly useless
Case 1: γ' is useful	
γ is useless	by definition
Case 2: γ' is RA useless	
γ is RA useless	since $\gamma_{ac}=\gamma'_{ac} \wedge \gamma_w=\gamma'_w$
Case 3: γ' is RA-useful and IA-useless	
$\exists \gamma'' \in v_{ia} \mid \gamma''_{ac}= \gamma'_{ac} \wedge \gamma''_w=\gamma'_w \wedge \gamma'_r>\gamma''_r$	definition of IA-useless
γ is useless	definition of IA-useless by γ''
QED for IAT1	
Case 1: $\gamma_w \geq \gamma'_r$	
γ is useless	RAT, γ' applied to (γ_{ac},γ_w)
Case 2: $\gamma_w < \gamma'_r$	
Let $\gamma''=(\gamma_{ac},\gamma_w,\gamma'_r)$	
$\gamma''_{ac}= \gamma_{ac} \wedge \gamma''_w=\gamma_w \wedge \gamma_r>\gamma'_r=\gamma''_r$	
γ is useless	IAT1 on γ''
QED	
Corollary: $\exists\ \gamma' \in v_{ia} \mid \gamma_w<\gamma'_r < \gamma_r$ $\Rightarrow \gamma$ is useless	
Let $\gamma''=(\gamma_r,\gamma_w,\gamma'_r)$	a valid AT
γ is useless	by applying IAT on γ''

Theorem 8.2: Proof of IAT Theorem

A *useless* AT can now be defined as one which has either a useless RA or useless IA. Useless ATs can be ignored. If one is IA-useless, another invalidate will preserve coherence for this AT. If one is RA-useless, then its reuse is already lost by virtue of invalidation. This preserves coherence too. It is not necessary to even worry about invalidates in its IA.

Useless ATs, once eliminated, prescribe no `INV`s at all. Thus, RAs cut by useless ATs are not useless by virtue of that particular AT. Similar comments apply to IAs. Fortunately, it is not necessary to untangle this mess because a stronger result holds. An RA which *covers* any AT (useful or not) is useless . A similar result holds for IAs:

RAT (theorem 8.1): Any RA which covers an AT is useless.
IAT (theorem 8.2): Any IA which covers another IA is useless.

The essence of the proof of RAT is that if an RA covers an AT, there is always a smaller AT that is useful and starts with the RA thereby directly killing it. Some examples of the particular cases of the proof of theorem 8.1 and shown in figure 8.8. The examples are merely indicative and not exhaustive of the possibilities. The reasoning for IAT and related theorems is similar.

At the end of an AT, α, any RA, β, that started before or with α, and hasn't ended, can be ignored. Similarly, any another AT α' that started before α can be eliminated. Any AT for which the essential write has already occurred can be eliminated. More precisely:

For a given AT $\alpha \in v_{at}$:

- An RA $\beta \in v_{ra} \mid \beta_{ac} \leq \alpha_{ac} \wedge \beta_{re} \geq \alpha_r$ can be eliminated as useless
- An AT $\alpha' \in v_{at} \mid \alpha'_{ac} \leq \alpha_{ac} \wedge \alpha'_r > \alpha_r$ can be eliminated as useless
- An AT $\alpha' \in v_{at} \mid \alpha'_w < \alpha_r < \alpha'_r$ can be eliminated as useless

The second rule subsumes the first one for RAs that start an AT, but the first one is still useful as an early elimination step. The third catches some cases the second does not (and vice-versa). These three theorems form a useful basis for building ATs and keeping the size of the final graph small. Applying the first and third at each node is sufficient.

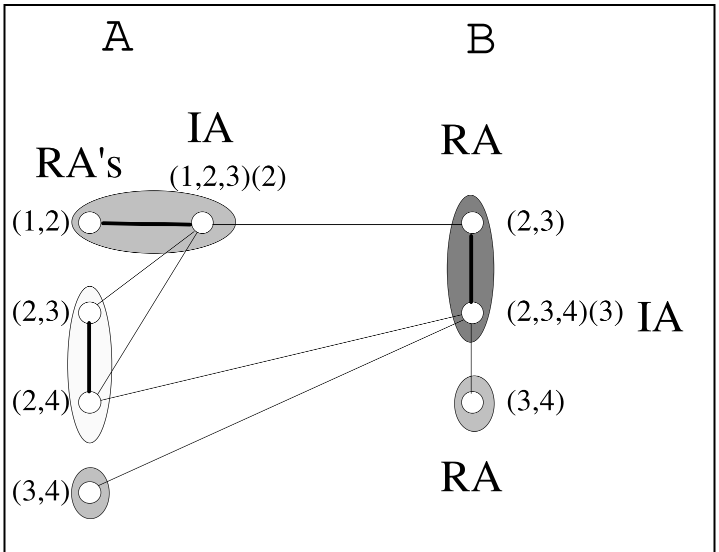


Figure 8.9 Coloring graph for figure 8.6

The RAT theorem is also sufficient. Any contradiction in the coloring graph will be found as a useless RA by RAT and eliminated (theorem 8.3). However, this does not say that RAT reduces the coloring graph to a minimal size. There may still be edges which can be eliminated without affecting the final coloring. The RAT and IAT reduction of figure 8.6 is shown in figure 8.9. It requires three colors to preserve all reuse.

8.3.2 Reducing the Coloring Graph

Further reductions in the epoch graph would remove information which is relevant to the original problem. This is particularly true for those cases where there are insufficient colors to preserve all reuse. Weighting decisions must be made. For instance, two RAs

$\exists\gamma\in v_{ia},\beta\in v_{ra} \mid \mathrm{color}(\gamma)=\mathrm{color}(\beta)\wedge\mathrm{color}(\gamma)\neq\mathrm{color}(\beta)$	an impossible coloring
$\exists i\in\bar{\gamma}$, $\mathrm{color}(\beta)=\mathrm{color}(v_\gamma(i))$	meaning of color(γ)=color(β)
$\forall i'\in\bar{\gamma}$, $(v_\gamma(i'),\beta)\in$ Edges	meaning of color(γ)≠color(β)
$(v_\gamma(i),\beta)\in$ Edges	in particular
$i\in\beta$	by definition of edges
$\beta_{ac}=\gamma_{ac}$	since color(γ)=color(β)
β covers $\alpha=(\gamma_{ac},\gamma_w,i)$	
β is useless by RAT, QED	

Theorem 8.3: Proof that RAT is sufficient

spanning exactly the same nodes would have a different effect than only doing so, in this case.

However, there are still some aspects of the coloring graph which are redundant. Even though they reflect real effects in the program they will not affect the outcome of the coloring decision. These results rest upon the nodes in the coloring graph, the IAs and the RAs, rather than the complete ATs. The equivalent IA theorem (theorem 8.4 & figure 8.10) states that if two IAs start and end on the same nodes (with different essential writes) only one of them need be considered. Such IAs will also be referred to as *useless*, though the sense is slightly different than for ATs. This result plus IAT (theorem 8.2) shows that at most one IA (for each variable) can start at a given node (theorem 8.5). That bounds the size of the coloring graph:

- $|v_{ia}| \leq |v_r|$ – Size of interference graph is linear

This does not constrain ATs, only IAs. Thus, the number of RAs is not likewise limited. But, in terms of driving the actual coloring, only $|v_{ia}|$ matters in any critical way (§8.5).

There are some RAs which are not the first half of an AT. This happens for the last reuse arc of any variable, among other places. It does not necessarily happen for the SA of every AT. Such RAs are said to be *dangling*. For such an RA, β, β_{ac} never needs to be invalidated. Even if β_{re} were a write that makes β_{ac} stale, β_{ac} will never be accessed again.

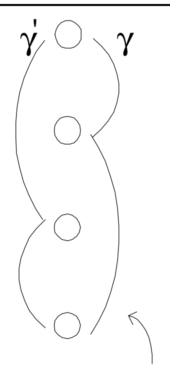


Figure 8.10: Redundant IA

There are two ways that an RA can dangle. It might start at the same point as an AT (bound) or not (unbound). Any unbound RA must be dangling, as a matter of definition. Reuse for unbound RAs can be preserved simply by having their essential access be a normal reference (cached and available in shared address space) not subject to the coherence strategy. Thus, they require no color at all and can be ignored in the coloring algorithm. Previous examples included them for sake of explanation of the principle.

Bound, dangling RAs must have the same color as the essential access of the AT to which they are bound. An important property of such RAs is that they cannot change the coloring of the graph (theorem 8.7). All this means is that β, bound to α, is covered by α_{ra}. Preserving reuse of α_{ra} also preserves it for β.

$\exists\, \gamma' \mid \gamma_{ac}=\gamma'_{ac} \wedge \gamma_r=\gamma'_r \wedge \gamma_w < \gamma'_w$ $\Rightarrow \gamma$ is redundant	Statement of theorem
Let Nodes' = Nodes - γ	Build up from the graph without γ
Let color' be an optimal coloring of Nodes'	
$\exists i \mid \text{color}'(v_{(\gamma'_{ac},\gamma'_w)}) = \text{color}'(v_{\gamma'}(i))$	since color' is valid
Let $\text{color}(v_{(\gamma_{ac},\gamma_w)}) = \text{color}'(v_{(\gamma'_{ac},\gamma'_w)})$	since both RAs start in same epoch
$\overline{\gamma'} = \overline{\gamma}$	since $\gamma_{ac}=\gamma'_{ac} \wedge \gamma_r=\gamma'_r$
Let k=i, $k \in \overline{\gamma}$	from previous step
let $\text{color}(v_\gamma(k)) = \text{color}(v_{\gamma'}(i))$	
$\forall j \in \overline{\gamma} \mid j \neq k$, let $\text{color}(v_\gamma(j)) = 0$	Definition of preserving coherence
$\forall (x,y) \in$ Edges, $x=v_\gamma(k)$	
$\Rightarrow \text{color}(x)=\text{color}'(v_{\gamma'}(i))$	
$\text{color}'(v_{\gamma'}(i)) \neq \text{color}(y)$	since color' is correct
$\text{color}(x) \neq \text{color}(y)$	all reuse preserved is preserved in color
color$\equiv$color'	Color' correctly handles γ as is
QED	Color' is optimal on the complete graph

Theorem 8.4: Equivalent IAs

$\neg\exists\, \gamma,\gamma' \in v_{ia} \mid \gamma_{ac} = \gamma'_{ac}$, not redundant and useful	statement of theorem
Case 1: $\gamma_r = \gamma'_r$	Redundant by 8.4
Case 2: $\gamma_r \neq \gamma'_r$	
wolog, $\gamma_r < \gamma'_r$	
γ' is useless	by IAT
QED	At most one IA starts at a given node

Theorem 8.5: Unique IA

Dangling RAs, bound or not, can be eliminated for purposes of finding the chromatic number of the coloring graph. Unbound RAs can be eliminated even from the process of finding a good coloring (where spills are needed). Bound RAs must be considered in this case though.

A corollary is that RAs which are bound by their reuse node (instead of their access node) to the essential write of an AT do not affect the chromatic number either (theorem 8.8). The essential reason is that the subset RA can be colored the same way as the super-setting RA without really adding any new edges.

$\neg\exists\ \gamma,\gamma' \in v_{ia}$, useful $\mid \gamma\neq\gamma' \wedge \gamma_{ac} = \gamma'_{ac}$	Statement of theorem
Assume that such a pair exists	
$\gamma'_r \neq \gamma'_r$	by Equivalent IA (theorem 8.4)
wolog, $\gamma'_r > \gamma_r$	
case 1: $\gamma_r \leq \gamma'_w$	γ' is useless by RAT (theorem 8.1) on γ
case 2: $\gamma_r > \gamma'_w$	
Let $\gamma''=(\gamma'_{ac}, \gamma'_w,\gamma_r)$	γ'' may be useless, but it doesn't matter
γ' is useless	by applying γ'' as subset of γ' under IAT
QED	

Theorem 8.6: One IA per variable per node

$\beta\in v_{ra}$ dangling $\equiv \neg\exists\alpha'\in v_{at} \mid$ $\alpha'_{ac}=\beta_{ac} \wedge \alpha'_w=\beta_{re}$	Definition of dangling
$\exists\alpha \in v_{at} \mid \alpha_{ac}=\beta_{ac}$	Definition of bound RA
$\beta_{re} < \alpha_r$	By RAT
$\alpha_w \neq \beta_{re}$	by definition of bound & dangling
Assume $\alpha_w < \beta_{re} < \alpha_r$	
case 1: $\beta_{re} \in v_r$	
$(\beta_{ac},\alpha_w,\beta_{re})$ is an AT	by definition of AT
$\rightarrow\leftarrow$	Cannot be by RAT
case 2: $\beta_{re} \in v_w\text{-}v_r$	
$(\beta_{ac},\beta_{re},\alpha_r)$ is an AT	by definition of AT
$\rightarrow\leftarrow$	Contradiction to definition of dangling
$\beta_{ac} < \beta_{re} < \alpha_w$	from previous three steps
color (β) = color (α_{ra})	since β is bound
$\forall\ v_\gamma(i) \mid i \in \overline{\beta}$	possible edges involving β
$i \in \overline{\alpha_{ra}}$	since $\beta_{re} < \alpha_w$
$(v_\gamma(i),\alpha_{ra}) \in$ Edges	definition of edges
$\text{color}(v_\gamma(i))\neq\text{color}(\alpha_{ra})$	from definition of Edges
$\text{color}(v_\gamma(i))\neq\text{color}(\beta)$	since color (β) = color (α_{ra})
$(v_\gamma(i),\beta)$ is a redundant edge	
QED	

Theorem 8.7: Bound, dangling RAs cannot change coloring

$\exists\alpha \mid \alpha'_{sa}=\alpha_{sa} \wedge \alpha_{ac} < \alpha'_{ac} \Rightarrow \alpha'$ is redundant	statement of theorem
Let Nodes' = Nodes-$\left(\bigcup_{i\in\overline{\alpha'_{ia}}} \alpha'_{ia}(i)\right)\bigcup\{\alpha'_{ra}\}$	All Nodes except those from α'
Let color' be a minimal coloring on Nodes'	
$\alpha'_{ia}=\alpha_{ia}$	by RAT
$\forall i\in\overline{\alpha_{ia}}$, let color($\alpha'_{ia}(i)$)=color($\alpha_{ia}(i)$)	
$\exists i\in\overline{\alpha_{ia}} \mid$ color(α_{ac},α_{ra}) = color ($\alpha_{ia}(i)$)	Definition of preserving coherence
color(α'_{ra})=color($\alpha'_{ia}(i)$)	Def. of coherence & transitive
=color($\alpha_{ia}(i)$)=color(α_{ra})	
If this is an impossible coloring graph:	A contradiction in the coloring
$\exists$(x,y)$\in$Edges $\mid$ color(x)=color(y)	must be from an edge $\notin$ Nodes'
i.e. in some final coloring, α' creates a conflict	
case 1: $\exists i \mid x=\alpha'_{ra} \wedge y=\alpha_{ia}(i)$	Edges involving α
$i\in\overline{\alpha'_{ra}}$	Definition of edges
$\alpha_w < i < \alpha'_w$	Definition of AT
$\rightarrow\leftarrow$	$\alpha'_{ia}=\alpha_{ia}$
case 2: $\exists I \mid x=\alpha'_{ia}(i) \wedge y=\alpha_{ra}$	similar to case 1
case 3: $\exists i,\alpha''\neq\alpha \mid x=\alpha'_{ra} \wedge y=\alpha''_{ia}(i)$	Edges not involving α
$i\in\overline{\alpha'_{ra}}$	Definition of Edges
$\alpha'_{ra}\subseteq\alpha_{ra}$	Hypothesis
$i\in\overline{\alpha_{ra}}$	
(α_{ra},y)$\in$Edges	Definition of Edges
color(y)$\neq$color(α_{ra})	Color' is correct
color(y)$\neq$color(α'_{ra})	since color(α'_{ra})=color(α_{ra})
$\rightarrow\leftarrow$ to assumption	that color(x)=color(y)
case 4: $\exists i,\alpha''\neq\alpha \mid x=\alpha'_{ia}(i) \wedge y=\alpha''_{ra}$	
$i\in\overline{\alpha''_{ra}}$	Definition of Edges
($\alpha_{ia}(i)$,y) $\in$Edges	Definition of Edges
color(y)$\neq$color($\alpha_{ia}(i)$)	Color' is correct
color(y)$\neq$color($\alpha'_{ia}(i)$)	since color(α'_{ia})=color(α_{ia})
$\rightarrow\leftarrow$ to assumption	that color(x)=color(y)
Coloring α' does not produce a contradiction	
$\lvert$color$\rvert$ = $\lvert$color'$\rvert$	coloring of α' does not add colors
color is a minimal coloring of Nodes	QED

Theorem 8.8: Subset RAs of the same AT do not change coloring

8.3.3 Loops

Outer serial loops are handled by unrolling them and treating them as sequential code. If the reference pattern is constant, it is sufficient to unroll the loop five times to find the best available cyclic coloring.

With respect to TS', loops of interest are of this form:

```
DO I
   PDO J
      v_ac
   PDO J
      v_ac
   ...
```

Everything that happens inside a `PDO` is summarized as a single reference, where the variable is either read, written, or both. It does not matter what other loops are inside of the `PDO` or how the reference pattern changes based on the parallel loop index, `J`. A constant reference pattern is one where the same sequence of references happens for each iteration of the `I` loop (table 8.2). This also assumes that whatever section or array that v represents is the same section across iterations of the `I` loop. This is a reasonably common case since it requires only that array indices do not vary based on the outer loop. For instance, a time-step loop around an inner parallel work loop is of this form.

$\theta 1$	etc., a reference type, either 'r', 'w', or 'rw'
$v^{i}_{\theta 1}$	v is accessed with type $\theta 1$ on `I` loop iteration i
$e_{k1}{:}v^{i}_{\theta 1}$	the access occurs on the k1'st epoch within the `I` loop (k's are not ordered)
$e_{k1}{:}v^{i}_{\theta 1} \Leftrightarrow e_{k1}{:}v^{i+1}_{\theta 1}$	constant reference pattern
$e_{k1}{:}v^{i}_{ia}$	IA starting in e_{k1} on iteration i, this uniquely identifies an IA (theorem 8.5), an IA may span epochs
$e_{k1}{:}v^{i}_{ia} \Leftrightarrow e_{k1}{:}v^{i+1}_{ia}$	constant IA pattern
$e_{k1}{:}v^{1}_{ia}$	a *pattern master* IA (for loops starting at 1)
$e_{k1}{:}v^{i}_{ia}$, $i>1$	a *pattern duplicate*

Table 8.2: Loop Definitions

It follows from the definition of an IA that if the reference pattern is constant, the set of all IAs (useful & useless) is a constant pattern. By itself this is not sufficient because it says nothing about how those IAs might interact. What is needed is that the pattern of useful IAs is constant:

Theorem 8.9: Unroll Stability — useful IAs form a constant IA pattern

Informal proof:
Redundant IAs can be ignored.
The theorem is equivalent to: $e_{k1}{:}v^{i}_{ia}$ is useless $\Leftrightarrow e_{k1}{:}v^{i+1}_{ia}$ is useless
assume $e_{k1}{:}v^{i}_{ia}$ is useless
$e_{k1}{:}v^{i}_{ia} = (e_{k1}{:}v^{i}_{ac}, e_{k2}{:}v^{i'}_{w}, e_{k3}{:}v^{i''}_{r})$
$e_{k1}{:}v^{i+1}_{ia} = (e_{k1}{:}v^{i+1}_{\theta 1}, e_{k2}{:}v^{i'+1}_{\theta 2}, e_{k3}{:}v^{i''+1}_{\theta 3})$
$\exists\, \gamma^{i} = (e_{k4}{:}v^{i}_{ac}, e_{k5}{:}v^{i'}_{w}, e_{k6}{:}v^{i''}_{r})$ which makes $e_{k1}{:}v^{i}_{ia}$ useless, by assumption
$e_{k1}{:}v^{i}_{ac} \le e_{k4}{:}v^{i}_{ac} \wedge e_{k3}{:}v^{i''}_{r} > e_{k6}{:}v^{i''}_{r}$ by IAT
$\gamma^{i+1} = (e_{k4}{:}v^{i+1}_{ac}, e_{k5}{:}v^{i'+1}_{w}, e_{k6}{:}v^{i''+1}_{r})$
exists, but is possibly useless, since accesses are a constant pattern
$e_{k1}{:}v^{i+1}_{ac} \le e_{k4}{:}v^{i+1}_{ac} \wedge e_{k3}{:}v^{i''+1}_{r} > e_{k6}{:}v^{i''+1}_{r}$
$e_{k1}{:}v^{i+1}_{ia}$ is useless by γ^{i+1}
The converse is similar.

Similar reasoning can be applied to the stability of RAs.

The key to making this work is that γ^{i+1} does not need to be useful itself. Merely that the IA exists is sufficient to insure useful-IA unroll stability.

Since the IA and RA pattern is constant, there should be some regularity to an optimal coloring as well. A constant coloring pattern (the same for every iteration) is not adequate, however, a cyclic one seems to be optimal. In all cases we have examined, it is optimal and it seems likely to be in general.

Definition 8.10: Cyclic Coloring — A coloring of a loop such that

color($e_{k1}{:}v^{i+1}_{\theta 1}$) = color($e_{k1}{:}v^{i}_{\theta 1}$)+*cycle_offset* mod *cycle_size*.
cycle_offset and *cycle_size* can vary by loop and are determined by the AT pattern.

For a cyclic coloring, the interference relationship of every IA stays the same in subsequent epochs. If γ^{i} cuts some RA β^{i} because they are the same color, then γ^{i+1} will cut β^{i+1} as well because γ and β both have their color changed by the same *cycle_offset*. Given theorem 8.9 and definition 8.10, the cost of particular coloring can be found by examining the effects of all IAs which begin in epoch 1 and considering all possible values of *cycle_offset* and *cycle_size*. This bounds the number of iterations a loop needs to be unrolled to find an optimal cyclic coloring:

> Theorem 8.11: Unrolling Five Times — Unrolling a loop five times is sufficient to generate an optimal cyclic coloring.
>
> Informal proof:
> It is only necessary to consider the effects of ATs beginning in epoch one.
> Such ATs must end by epoch three.
> Any RA which starts by epoch three can be definitely determined to be useful or not by the end of any contained ATs.
> Such ATs must end in at most two more epochs (past epoch three).

In many cases it will not be necessary to unroll five times. The more epochs there are in the loop the fewer unroll iterations that will probably be needed. The precise amount of unrolling depends on an exhaustive case analysis of read and write patterns.

After the loop is colored, it is patched back into the main coloring graph logically as a case-like construct with one branch for zero iterations, one branch for one iteration, plus additional two-iteration branches for each possible starting color (e.g. if a loop had a *cycle_size* of 4, there would be 6 total branches). This is sufficient to encompass all possibilities because any AT which interfaces the unrolled loop to the outside code can connect to at most the first two or the last two iterations. Every possible coloring combination on loop exit is exposed. In practice, there will usually not be any point to trying to preserve reuse on loop exit. Any color which was invalidated in the loop is invalidated upon loop exit. Entry conditions are handled by leaving the loop colored only as a set of color offsets and not any particular colors. The coloring of ATs external to the loop then forces the best starting color for the loop. This requires that the loop coloring be summarized only as a single two-iteration case.

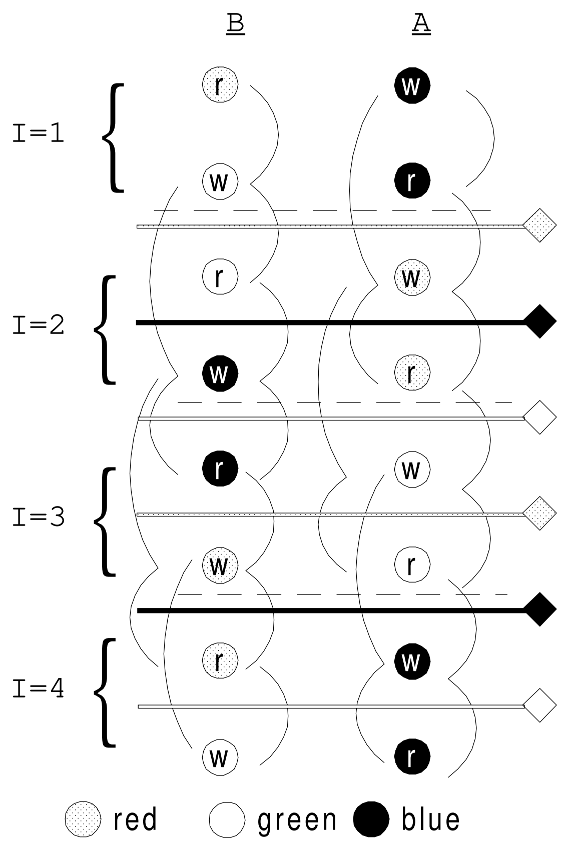


Figure 8.11: Heat Flow, Unrolled 4 Times

Example

The computation in a grid relaxation (e.g. a 2-D heat flow), reduced to its coherence essentials, is:

```
DO I
   PDO J
      A(J,*)=B(J,*)
   PDO J
      B(J,*)=A(J,*)
```

The reference pattern is constant for each iteration of the `I` loop and is simply:

```
A=B
B=A
```

The epoch graph, its final coloring, and invalidate placement is shown in figure 8.11. This corresponds to the following code:

```
DO I
 PDO J
    A(J,*) [blue] =B(J,*)  [red]
 INV green
 PDO I
    B(J,*)[green]=A(J,*)  [blue]
 INV red
 CycleColor(1)
```

Here `CycleColor(1)` means that on the next iteration of the `I` loop blue references will be red, green reference blue, and red references green. This also applies to the color of the invalidate.

A simple but common case is that of chains (§6.2). An important property of chains is that (in isolation) an optimal coloring is simply to alternate between two colors for each subsequent node.

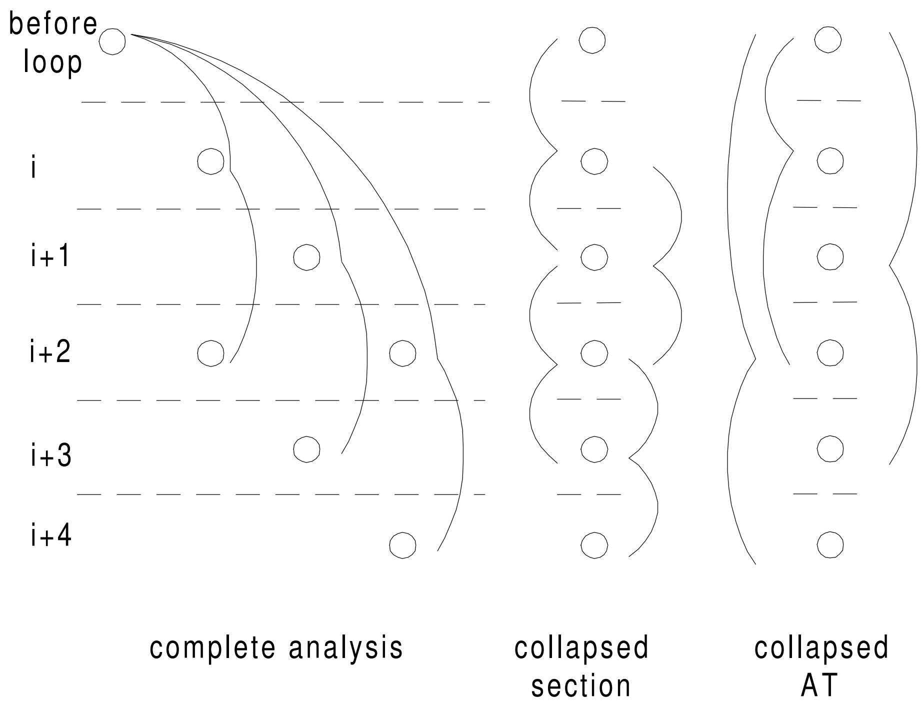


Figure 8.12: Non-Constant Loop References

Non-constant Reference Patterns

For those loop patterns which are not constant, the default behavior is to choose a conservative section such that the pattern does become constant. This does not mean the section analysis has to become more conservative. Consider the following case:

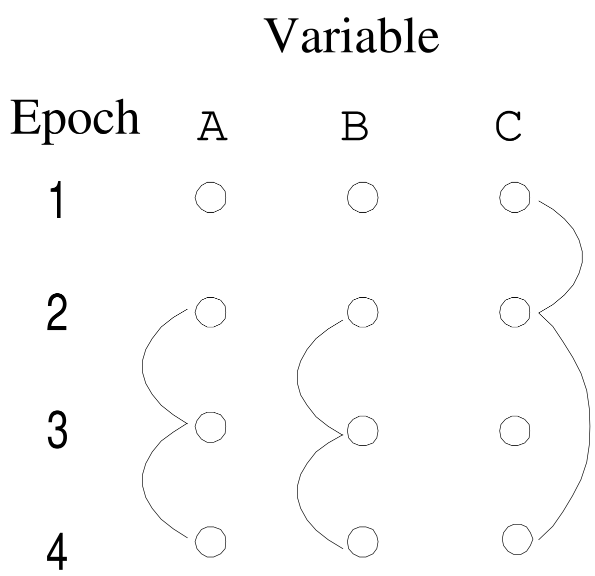
```
A()=
DO I
   PDO J
      A(I,J)=A(I-2,J)
```

The reference pattern is not constant because the section of `A` which gets referenced on each iteration of the `I` loop is different. If the analysis considered `A(I,J)` and `A(I-2,J)` to be in the same section, it would appear to be a chain (§6.2) and every reference would be invalidated two epochs after it is made which is before it is reused (as a coincidence it would do the right thing for a right hand side of `A(I-1,J)`). Instead, just the ATs are compressed to refer to the same section (figure 8.12). The RA pattern for each iteration is now constant (there are no ATs within the loop). Loop coloring leaves all reuse in place. When the loop is embedded in the complete program along with the initialization, a single invalidate is added after the initialization. That is the best that can be done with finite colors, so in this case the constant IA pattern approximation was as good as an optimal coloring. This will not hold in general.

8.3.4 Write reuse

As so far presented TS' assumes that there is reuse from a read epoch to a write epoch. If this is not so, an RA that goes from an access node to a write (only) node is not worth preserving. Such values are *reuse-dead* because they must be invalidated before they can be reused. There is no point in delaying the `INV`, so far as reuse is concerned. These values can be removed from cache at the end of the essential access epoch. This is not strict read-through and write-through because it is important (in general) to keep the values in cache until the end of the epoch in order to preserve intra-epoch reuse. If reuse-dead values are immediately eliminated, any AT which was a write only node as its essential write can be eliminated. However, it does not remove any ATs for which the essential access node is the essential write of a different AT.

The apparent implementation of this is a special marking bit for reuse-dead accesses. At the end of an each epoch, all reuse-dead accesses can be invalidated. This mechanism is just the same as that already presented for TS' where reuse-dead accesses are marked as being in a special region, the *null region*. With this in mind, it is worth examining the question of where the invalidate should go for null regions. Instead of considering null regions as a special case to be handled immediately, it is actually better to simply consider reuse-dead accesses like all others, but with the RA following the null region *converted* to an IA. This will reduce the number of regions that are needed.




```
PDO
   =C(I)
ENDDO
PDO
   C(I)=C(I)+B(I)+A(I)
ENDDO
PDO
   A(I)=I
   B(I)=B(I)+1
ENDDO
PDO
   =C(I)+B(I)+A(I)
ENDDO
```

Figure 8.13: Epoch Graph for Null Region Invalidation

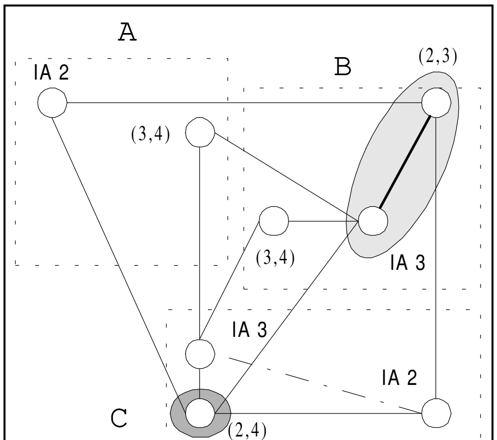


Figure 8.14: Coloring Graph for Figure 8.13 with conversion

Consider the program in figure 8.13. The reads of `A` in epoch 2 are reuse-dead because `A` is written (but not read) in epoch 3. Neither `B` nor `C` are reuse-dead because they read in the same nodes on which they are written. Since $\mathtt{A}_{ac}$ is reuse-dead, it could be invalidated after node 2 (instead of node 3 as would be the case if it were not reuse-dead). The coloring graph that results from this is shown in figure 8.14. Epoch 2 of B is given color 1. The invalidate at epoch 3 must also be color 1 to preserve correctness. Since the invalidate at epoch 3 cuts the $\mathtt{C}_{(2,4)}$ reuse arc, Epoch 2 of `C` must be a different color, color 2. Now, an `INV` for the reuse-dead $\mathtt{A}_{ac}$ will destroy `B` reuse if it is color 1 and destroy `C` reuse if it is color 2. Therefore, three colors would be required if the reuse-dead accesses were immediately invalidated.

Instead, consider them to be normal accesses (figure 8.15). There is no node for $\mathtt{A}_{(2,3)}$ by the nature of $\mathtt{A}_{ac}$ being reuse-dead. Instead, $\mathtt{A}_{(2)}$ now appears as an IA node indicating that invalidation could occur between epoch 2 and 3. Also, the $\mathtt{A}_{(3)}$ IA resulting from $\mathtt{A}_{at}$

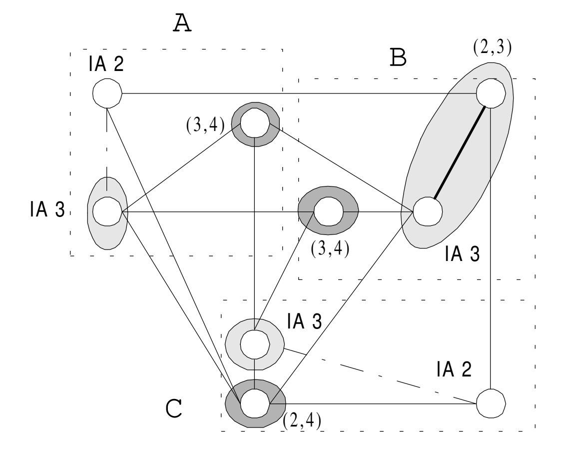


Figure 8.15: Coloring Graph for Figure 8.13 without conversion

is still in the graph because coherence still needs to preserved even though A_{ac} is reuse-dead. $A_{(3,4)}$ is unaffected and still appears as an RA. $A_{(2)}$ and $A_{(3)}$ are now alternates as if they were in the same IA. It is necessary to only choose one of them in order to preserve coherence. There is now a two coloring (figure 8.15).

Sometimes this conversion will save colors. It will never require more colors because it still always possible to choose the IA node that resulted from converting the reuse-dead RA node.

Read-through and Write-through

For those epochs which are reuse-dead and there is no intra-epoch reuse, read-through and write-through no-allocate (i.e. references which do not leave anything in cache) can be used as an optimization. These have the same effect as an `INV` immediately following the epoch. That is, no RAs or ATs start there. This needs no regions and leaves them all available.

8.3.5 Root Processor

As a partial exception to the assumption that schedules are not known, the root processor represents a special case. We will refer to it as P0 and all other processors as P1+. If P0 is assigned all of the serial work. Any RA that goes from one serial epoch to another (by jumping a parallel epoch in between) does not represent part of an AT. There is no chance of anything becoming stale because a write surely replaces the value currently in cache (definition 3.2 cannot be satisfied).

P0 also usually takes part of the parallel work. If this work is not distinguished, P0 must make every invalidate that P1+ does. It is difficult to extract additional benefit in this case. Therefore, it is necessary to consider P0 not part of P1+. This does not mean the user code needs to make a distinction or not share parallel work with P0, but only that coherence operations will be different on P0, even in a parallel region. Second, if P0 achieves a coloring that preserves all reuse, that coloring must surely work for P1+

(assuming P0 takes part in parallel work) even though part of it is irrelevant. But it may be possible that P1+ achieves an optimal coloring while P0 has to make a weighted decision and sacrifice some reuse or P0 may have more colors available to it.

In those cases where the distinction is worth making, AT analysis needs to be modified. ATs can move back and forth between P0 and P1+. There are only three patterns that form a true AT (definition 3.3):

1. $P0_{ac}$, $P1_w$, $P0_r$
2. $P1_{ac}$, $P1_w$, $P1_r$
3. $P1_{ac}$, $P0_w$, $P1_r$

Case 1 strictly belongs to P0. Cases 2 & 3 strictly belong to P1+. With that separation it is possible to solve both as completely independent problems. The simple reason is that the essential access and the essential read both occur on the same processor (set). Thus, the invalidate decision and coloring decision both apply to the same processor. Given this separation, the next issue is how to preserve reuse on RAs (or SAs) that cross between P0 and P1+ (since each end is now in a separate problem). This is a non-problem because there can be no such reuse. Such arcs simply appear nowhere in either coloring graph. The effect will be that such reuse appears as an RA in both graphs, naturally, without making any special cases. All of the constructions theorems (8.1, 8.2, & 8.4) go through as before even though the essential write is not part of the coloring graph for some ATs (keeping in mind that an AT owned by P0 has no elimination effect on P1+ even though the essential write happens on P1+).

8.4 NP-Completeness of TS' with control flow

As so far presented, TS' has been applied only to sequential code at the epoch level, i.e. there is no control flow between parallel loops. The definitions extend naturally to different control flow paths. Each possible path between the three nodes of an AT form a different AT. Interference (edges in the coloring graph) exists where some write, v_w, is in the range of some RA, u_{ra}. Note that u_{ra} may cross multiple basic blocks and v_w interferes only where its AT shares a path (or part thereof).

```
PDO
   A(I)=C(I)
ENDDO
PDO
   C(I)=
ENDDO
IF (c)
   PDO
      A(I)=A(I)+1
   ENDDO
ELSE
   PDO
       B(I)=C(I)
   ENDDO
ENDIF
PDO
   B(I)=
ENDDO
PDO
   =A(I)+B(I)
ENDDO
```

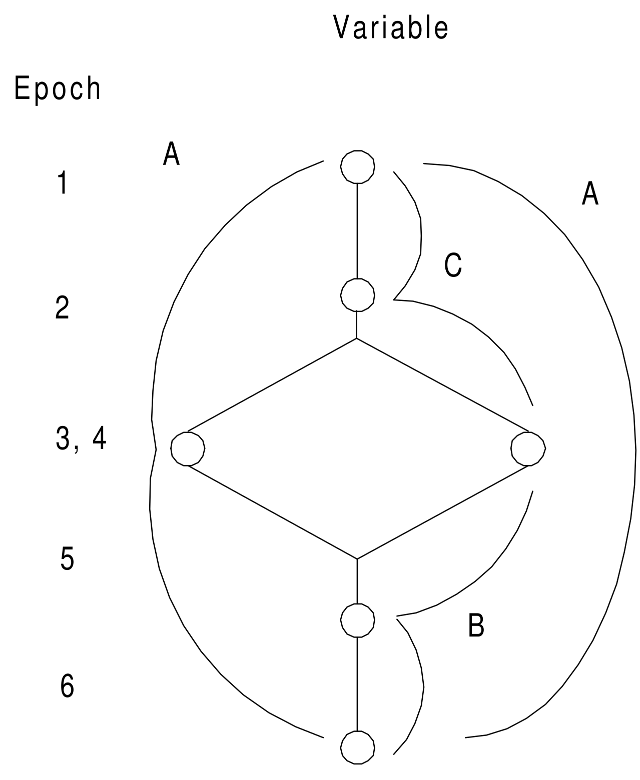


Figure 8.17: ATs over control flow paths

For instance, in figure 8.17, A_{at} flows along the left control path while B_{at} and C_{at} flow along the right path. There is also a dangling RA for A on the right control path because the reference to A in epoch 1 could be reused in 6 without necessarily being made stale by the write in epoch 3. Neither theorem 8.1 (RAT) nor theorem 8.7 immediately apply here because different paths are involved.

In the presence of control flow there are more choices for the placement of an invalidate. For instance, even though C_{ia} spans only a single epoch, the invalidate for it could be either before the fork or on the right branch. Only the one before the fork cuts A_{ra}. The choices will be represented by a path from the appropriate epoch. 'r', 'l', 'j', & 'p' represent right path, left path, join, and primary (not yet out of the same basic block). C_{ia}'s invalidates are represented C_{ia}(p) and C_{ia}(r). A_{ia}'s invalidates are represented by A_{ia}(3p), A_{ia}(3j), and A_{ia}(5). The coloring graph for figure 8.17 is show in figure 8.18. The A_{ra} along the right control flow path is represented by the single $A_{(1,6)}$ node. For illustration purposes the coloring graph is incomplete because it does not represent that both A's must have the same color.

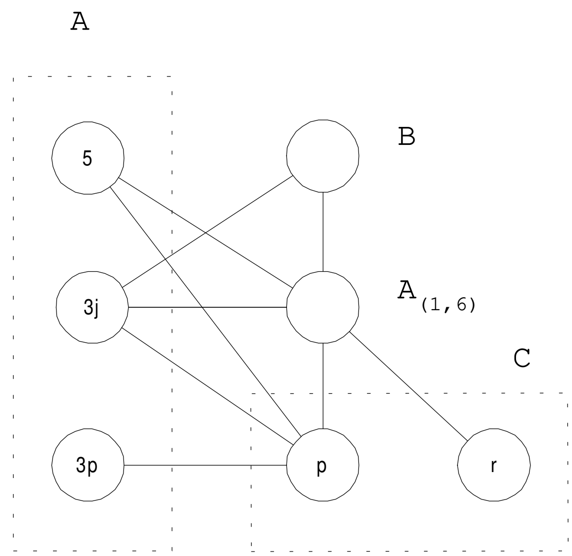


Figure 8.18: Coloring Graph for Figure 8.17

Simple fork-join control flow without the presence of loops, weighting, or same-color constraints is sufficient to make the coherence coloring problem NP-complete. The reduction is from 3-SAT [26].

```
(PDO's  compressed to one line)
A1=A1+1
B1=B1+1
C1=C1+1
F1=F1+1
IF (c1)
   A1=A2
   A2=A1+A2
ELSE
   IF (c2)
      B1=B2
      B2=B1+B2
   ELSE
      IF (c3)
         C1=C2
         C2=C1+C2
      ELSE
         F1=F2
         F2=F1+F2
         F2=F2+1
      ENDIF
      C2,F2=F2+C2
   ENDIF
   C2,B2=C2+B2
ENDIF
B2,A2=B2+A2
C2=F2
=A2,B2,C2
```

Figure 8.19: *3-gate* Code

Theorem 8.12: Coloring *Coloring Graphs* is NP-Complete

Proof: It is trivially in NP for the same reason that graph coloring is. The construction varies only in detail. The proof that 3-SAT can be reduced to 2-coloring a coherence-coloring graph follows in detail.

To show that it is NP-hard each Boolean variable and its negation from the 3-SAT expression (represented by *w*, *x*, *y*, *z*) is converted into a program variable (represented `A`, `B`, `C`, `D` etc.). Additionally one special program variable '`false`' is added to the graph. Each 3-SAT variable has a *starter* control flow sub-graph that forces x and $\overline{x}$ to be opposites. Each factor in the 3-SAT expression is represented by a *3-gate* control flow subgraph. The gate graph is constructed in such a way that if `A`, `B`, `C`, `D` all have the same color immediately prior to the gate graph, the gate will require three colors. Otherwise, it will require only two. One of the four 'inputs' will be '`false`'. Thus, the 3-SAT expression can be satisfied if and only if the coherence-coloring graph is 2-colorable. A graph can be 2-colored in O(n) time but the coherence-coloring graph is not a graph in the technical sense. The choice of invalidate nodes makes it a different problem.

A code segment to produce a 3-gate is shown in figure 8.19 (each assignment statement is understood to be a `PDO` loop). The corresponding epoch node graph is shown in figure 8.20. The first four nodes, $A_1{}^1$, etc., are in the starter graph and shared among all 3-gate graphs; these are the givens. The other nodes are unique to each particular 3-gate and colored as appropriate.

A couple of simplifications are made before constructing the coherence-coloring graph.

The first one is to join the A_1 chain and A_2 chain into a single chain. If they had been the same variable there would have been an RA from $A_1{}^1$ to $A_2{}^4$ around all of the other `IF` branches. This would have unnecessarily complicated the construction. As it stands, $A_2{}^3$ and $A_1{}^1$ can be red (for instance) and $A_2{}^2$ and $A_2{}^4$ can be green in an optimal two coloring. Attempting to change the 'parity' by coloring $A_2{}^3$ green would force 3 colors to be used. Thus, the two chains can be joined for the purposes of coloring. The last two nodes of every chain never become stale and do not need to be explicitly colored (or named in a chain).

The second simplification is to ignore the zero length basic block following an `ENDIF` and preceding the immediately subsequent parallel epoch. An invalidate there can only cause additional interference problems as compared to leaving it on one of the join branches (if both join branches have the same color invalidate that is still the same cost as having a single invalidate after the `ENDIF`).

The essence of the epoch node graph is that $F_1{}^1$ is colored red (arbitrarily) and identified with '`false`'. For there to be 2-coloring, the penultimate segment of each chain must have the same color invalidate as the first node in the chain ($A_1{}^1$, etc.). The final segment of the F chain must have the opposite color of $F_1{}^1$, e.g. green. That invalidate must cut either the penultimate segment of the A_2, B_2, or C_2 chain. Thus one of A_1, B_1, or C_1 must be green (i.e. '`true`') to permit a 2-coloring. The if-then-else structure is used to keep F from interfering with other segments on the chains. If it were to do so, a 2-coloring would not be possible in some of the needed cases.

To prove that there are no 'trick' colorings that might achieve the same effect it is necessary to look at the coherence-coloring graph (figure 8.21). The numbers in the nodes are the invalidate possibilities corresponding to the node in the epoch node graph with the same superscripted number (for example, $A_2{}^4$ in the epoch graph has only one possible location for its invalidate which is represented by the single '4' node under A).

Theorem 8.13: 3-gates are 2-colorable if and only if all not all inputs have the same color.

Proof: by cases: In all cases, wolog let '`false`'^{1} ($F_1{}^1$) be red. Case 4 is drawn out in more detail (figure 8.22).

Case 1: Assume A^1, B^1, C^1, and F^1 all have the same color, red. Attempt a 2-coloring. Whichever '4' nodes are chosen must be green since they are in a chain that starts with red. No '4' node can interfere with any other to achieve a 2-coloring since they are all green. A^4 must be chosen. Therefore, F^{4c}, C^{4b}, and B^{4a} cannot be chosen. B^{4b} must be chosen as the only alternative for B. Now, F^{4b} and C^{4a} cannot be chosen since they interfere with B^{4b} . The only alternatives left are F^{4a} and C^{4c} which interfere with each other. Therefore, no 2-coloring is possible.

Case 2: Assume A^1 is green and F^1 red. The following 2-coloring works:
A^4 - red
F^{4c} - green
C^{4c} - chosen, any color
B^{4b} - chosen, any color

Case 3: Assume A^1 & F^1 are red, C^1 green. The following 2-coloring works:
A^4 - green
F^{4a} - green
C^{4b} - red
B^{4b} - chosen, any color

Case 4: Assume A^1, F^1, & C^1 are red, B^1 green. The following 2-coloring works:
A^4 - green
F^{4b} - green
C^{4a} - green
B^{4a} - red

QED

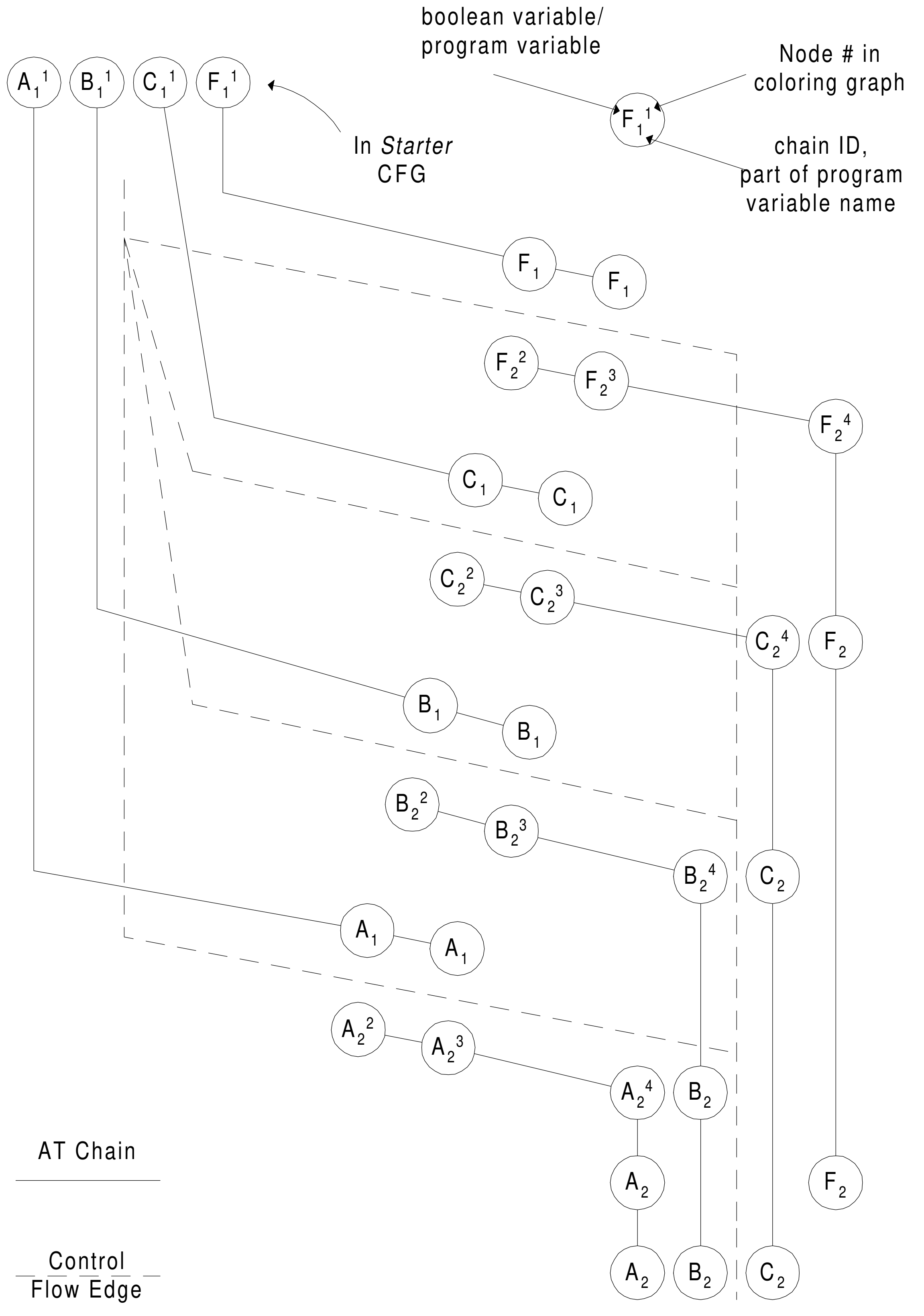


Figure 8.20: *3-gate* Control Flow Sub-Graph

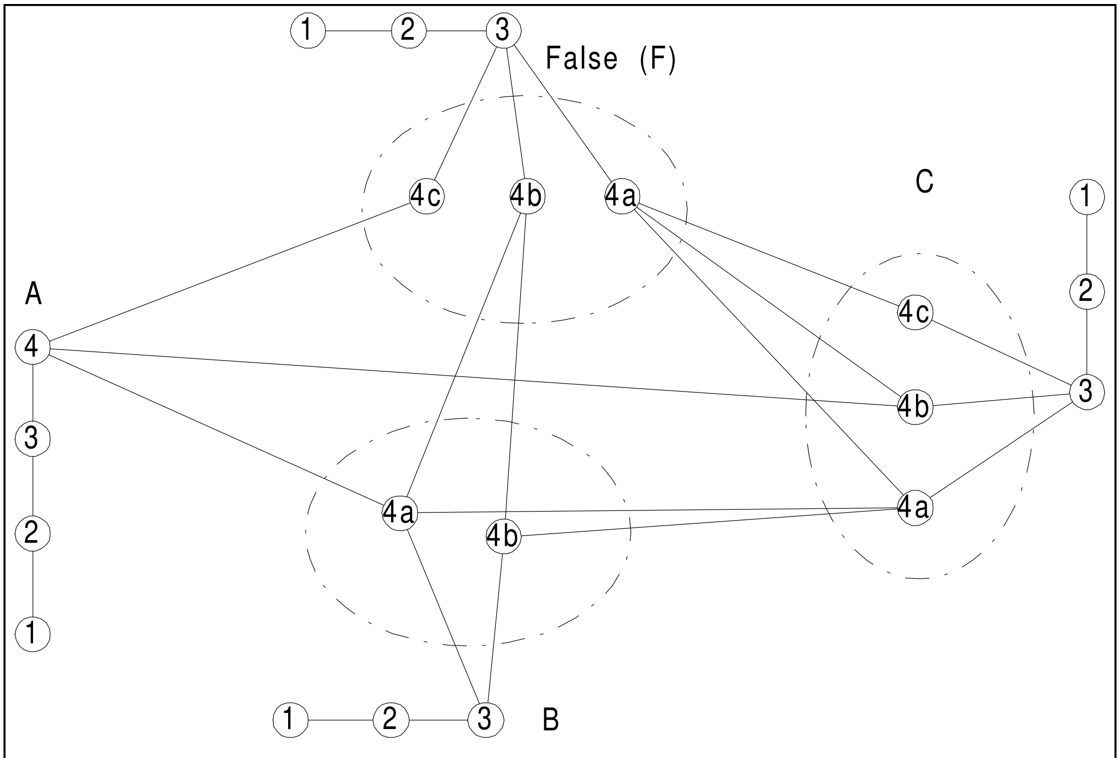


Figure 8.21: Coherence-Coloring Graph for Figure 8.20

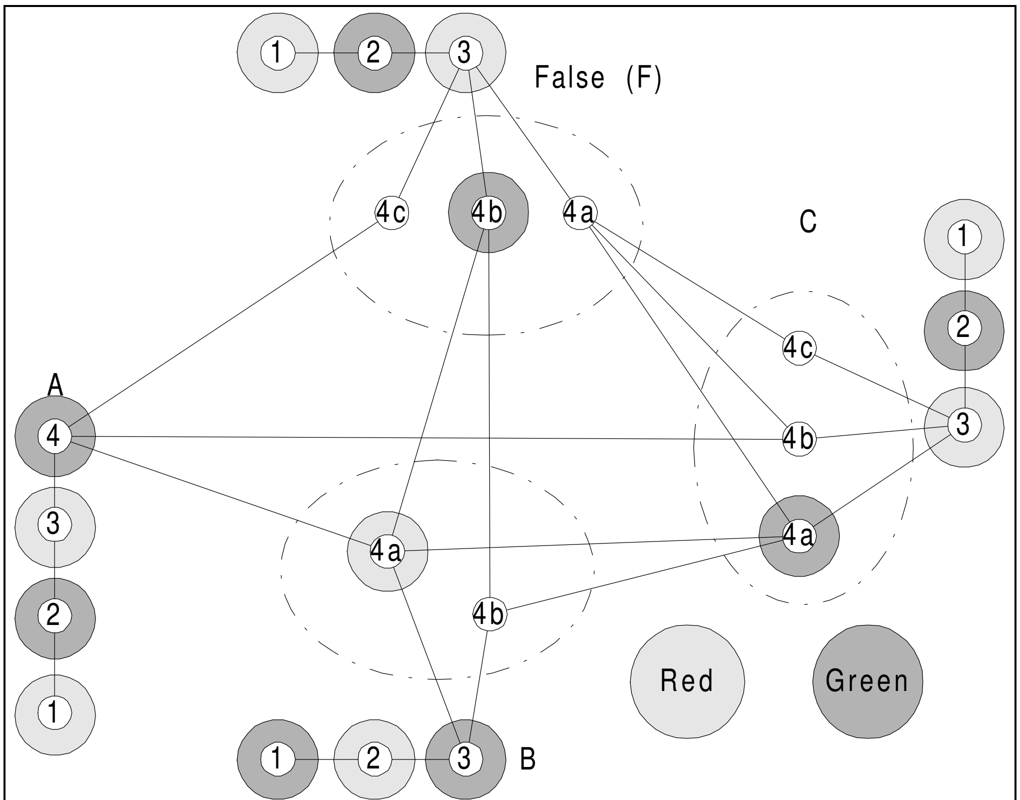


Figure 8.22: Case 4 of Theorem 8.13

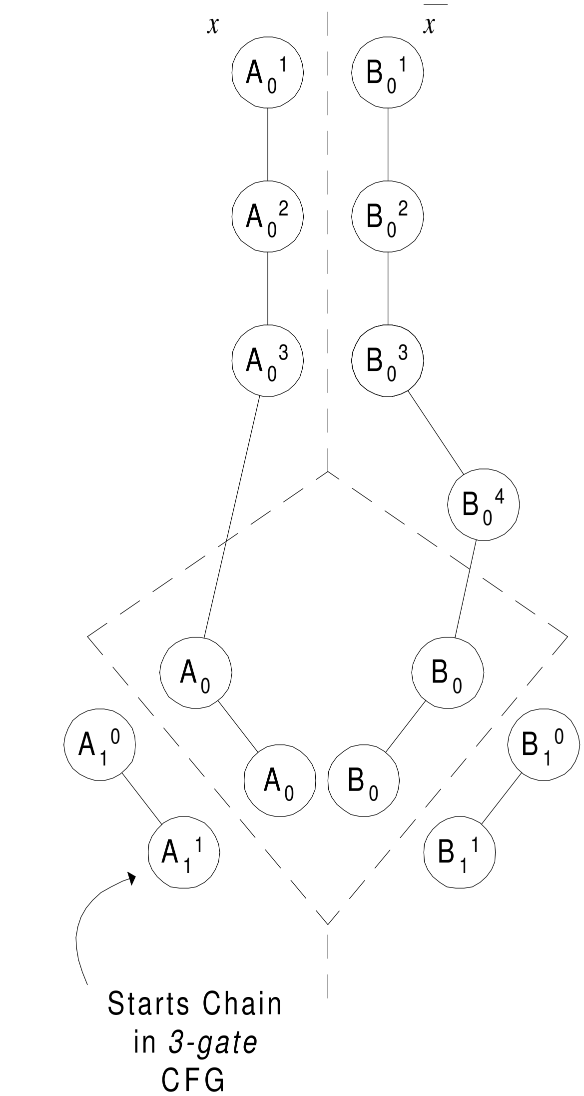


Figure 8.23: Starter Gate

The next component needed for a 3-SAT reduction is the starter gate (figure 8.23. For the starter gate to be 2-colorable, $A_1{}^1$ and $B_1{}^1$ must have the same color. Given that, $A_2{}^1$ and $B_2{}^1$ must have different colors. $A_1{}^1$ is identified with some Boolean variable in the 3-SAT expression, *x*. $B_1{}^1$ is then identified with its negation, $\overline{x}$. If a 2-coloring exists, *x* and $\overline{x}$ are guaranteed to have 'opposite' colors and thus a truth assignment is generated (by comparison to the color of the '`false`' node). The final step is to build a starter gate for every Boolean variable and connect their exit points. To that point the entry point for every 3-gate (for every factor) is connected. This completes the NP-completeness proof.

Figure 8.24 is a complete example for the expression $(x+y)(\overline{x}+y)(x+\overline{y})(\overline{x}+\overline{y})$. 2-SAT is used just to make the example small enough to understand. It cannot be satisfied and therefore should not have a 2-coloring. The sample assignment is *y* true, *x* false. The relevant invalidates are included in the figure (uninteresting ones are omitted). The problem comes in the 3rd factor (marked with '?') where the '`false`' chain needs a green invalidate in one of the two indicated places but either one of them would destroy reuse (on either A or D). The other invalidates are all in their final places preserving coherence and reuse. Deleting the 3rd factor would leave the graph with a valid 2-coloring indicating that *y* true, *x* false satisfies $\left(x+y\right)\left(\overline{x}+y\right)\left(\overline{x}+\overline{y}\right)$.

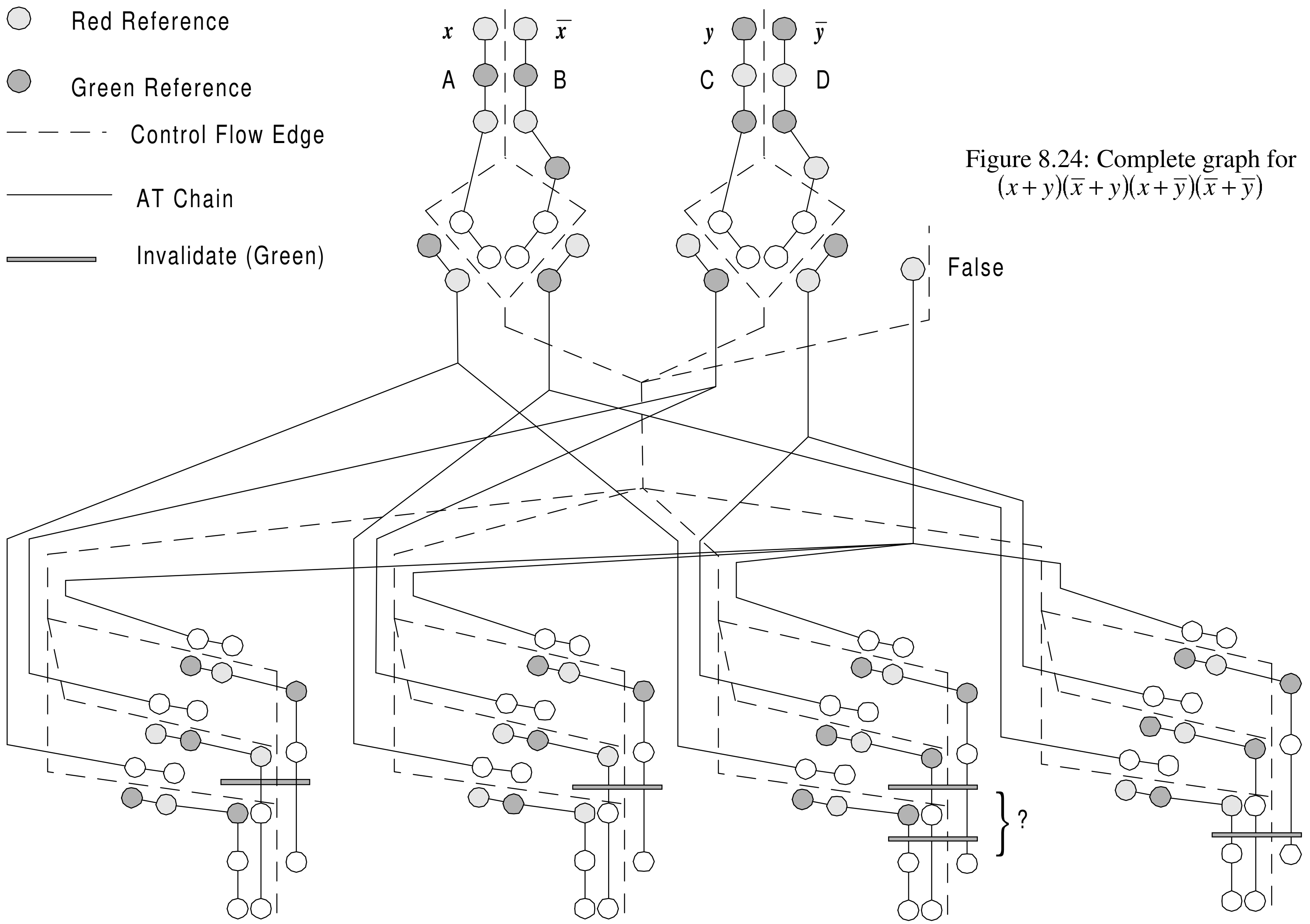


Figure 8.24: Complete graph for
$(x+y)(\overline{x}+y)(x+\overline{y})(\overline{x}+\overline{y})$

8.5 TS' Compiler Algorithms

With respect to TS', the compiler takes as input an inter-epoch control flow graph, a description of which sections are referenced in which epoch, the nature of that access (read, write, both, or neither). It has three tasks, constructing the ATs/epoch graph, coloring that graph, and mapping the result back to program code (we have implemented the first two, and all three are discussed in §9.1). The output is then a program augmented with invalidate instructions, region (run-time areas to which colors are applied, §8.6) declarations, and region to color mappings. The first two are discussed in the following sections. The discussion of mapping back to the program is deferred (§8.7.4) until after the hardware implementation is discussed.

Scalars (and very small array sections) which need coherence control are not assigned colors to avoid wasting a limited resource. Instead they are directly invalidated. While this may be slow, it is only applicable to a finite number of words and should not be an issue. One of the most common cases of sharing scalars across processors is when the root processor updates some value that is then used by all other processors. In this case, non-root processors can be executing the invalidate instructions during the serial epoch while the root processor is doing the computation. This should in most cases completely cover the cost of the address specific invalidate.

8.5.1 Building ATs for TS'

For sequential epochs, the ATs can be built in a single forward pass over the epochs, using sets of variables in a pseudo data-flow framework (figure 8.25)

Step 5 is simply the definition an AT. The '$\times$' means cross product restricted to identical variables, i.e. for every stale arc on v an AT will be constructed if v is read in this epoch. Step 7 is a direct implementation of RAT (theorem 8.1). The AT filter steps happens before Accesses_{live} is used because the condition in RAT is $\alpha_r \leq \beta_{re}$, not merely '<', for AT α to make RA β useless. Step 8 computes the actual RAs (and SAs) for later use in coloring the graph and not for building ATs. This particular definition captures all varieties of reuse including write to write.

Step 10 is a non-obvious implementation of IAT (theorem 8.2). Let $\beta \in \text{Stales}_{in}$ and $\alpha \in \text{ATs}$, both for the same variable v. If α_r is in the current epoch, then $\beta_{re} < \alpha_r$. Thus

1.	Reads ≡ variables read in this epoch
2.	Writes ≡ variables written in this epoch
3.	Stales ≡ variables which have been accessed, written, and will be stale if read again without invalidation
4.	ATs ≡ ATs (for all variables) which end in this epoch
5.	ATs = $\text{Stales}_{in} \times$ Reads
6.	Accesses_{local} = Reads + Writes
7.	$\text{Accesses}_{live} = \text{Accesses}_{in}$ - ATs
8.	RAs = $\text{Accesses}_{live} \times \text{Accesses}_{local}$
9.	$\text{Accesses}_{out} = \text{Accesses}_{live} + \text{Accesses}_{local}$
10.	$\text{Stales}_{out} = (\text{Stales}_{in}$ - ATs) + $\text{Accesses}_{live} \times$ Writes

Figure 8.25: TS' Data Flow

$(\beta_{ac},\beta_{re},\alpha_r)$ must be an AT, α'. For β to be used in Stales_{out} to produce subsequent ATs, there must be some later $v_r > \beta_{re}$. But in this case, α' would render $(\beta_{ac},\beta_{re},v_r)$ useless by IAT.

The Equivalent IA theorem (8.4) is implicit in the implementation of the cross product. Two members of Stales_{in} with the same essential access but different essential writes are considered to be a single member.

The complete lists of ATs and RAs are now scanned and moved from being owned by their last node to being owned by their first nodes. The ATs become IAs just by definition and the RAs stay as they are.

8.5.2 IAColor — TS' Coloring Algorithm

Since the coloring problem is NP-complete, there is not likely to be any 'best' way to approach it. The approach described here colors the graph by exhaustively searching all possibilities. There are several aspects of the topology that can be put to good use to narrow this search. For all of the programs that we have examined, the exhaustive IAColor completes in under one second (on a 486DX/33). It's implementation is approximately 2000 lines of C++ including heavy commenting, parsing (of a very simple 'language'), and AT building as well as the central coloring algorithm. There are surely better approaches but this one is workable.

The coloring graph is bipartite with IAs on one side and RAs on the other. Since invalidates have color and position (whereas RA nodes only have color), we view invalidate placement as the space to be searched. For any placement of invalidates, the cost of broken RAs can be computed (coloring invalidates implicitly colors RAs too). The essential algorithm is then to arrange the IAs in some order, try all invalidate/color combinations for each one, compute the cost, and pick the best (figure 8.26). The cost for a given IA is the cost imposed by all already chosen colors (nAboveCost) plus the cost of previous IAs cutting this one (since it just got colored) and the cost of this one cutting previous IAs which were already colored (nBranchCost). Each recursive call (to Color) returns the best total cost for coloring the graph for all possibilities below this node and a fixed coloring above.

If the specified number of colors available in the machine is sufficient for the program being colored, IAColor returns an ideal coloring with no excess invalidates, i.e. one which will correspond to an ideal local strategy. If there are insufficient colors, IAColor returns a lowest possible cost coloring. If each RA is assigned some cost, the net cost of those which are broken by invalidates is guaranteed to be the minimum possible. The current strategy simply counts each RA as costing 1. This could be any arbitrary function based on the properties of the RA itself (e.g. accounting for RA lengths or data volume). Cost functions cannot be based upon the current coloring beyond the simple question of whether or not the current RA is cut (we cannot imagine any such function which would be useful).

Expressed this simply, it would be impractically slow and explore a lot of recognizably dead branches. The implemented algorithm takes the following major steps, in order of importance, to reduce the search time:

1. A branch & bound approach is used to prune the search
2. The IA list is ordered in epoch node order, $\overline{\gamma}$ is searched backward
4. Only one new color is tried for each IA
5. The cutting cost is split into nCutsMe & nICut components

Starting with step 2, the IAs are put in a canonical order based on their starting node. The reason is that AT graphs tend to be 'skinny' in the sense that their connectivity is limited by the number of sections but their length is limited only by the number of epochs in the program. Arranging IAs in this order causes conflicts to occur earlier rather than later.

```
int Color(IA γ, int nAboveCost) {
    nMinCost = nAboveCost
    ∀ c∈ all_used_colors ∪ one_new_color {
        color γ with color c
        ∀ i∈ γ̄ {
                place an invalidate, INV,  at i with color c
                nBranchCost = nAboveCost + cost imposed by INV
                if (γ.next) return (Color(γ.next,nBranchCost))
                else {
                        if (nBranchCost < nBestSoFar) remember new best
                }
                nMinCost = min (nMinCost,nBranchCost)
        }
    }
    return (nMinCost)
}

Color (first γ)
```

Figure 8.26: IAColor, Outline

Thus, new colors are chosen based on as much information as possible (essentially a breadth first coloring of the graph). Let the IAs in this list be referred to as γ_1, ... γ_n.

Step 1, the branch and bound approach is probably the most important optimization. It is implemented by finding a lower bound based on the tail of the IA list. Before the optimal cost starting at γ_i is found, the optimal cost starting at γ_{i+1} is found. nAboveCost plus an optimal coloring of everything below lower bounds the cost at this node (no matter how it is colored). If this is greater than a known feasible solution, the whole branch can be pruned.

Step 3 tries the invalidate first at the end of the IA instead of at the beginning. At the beginning, it will surely conflict with the RA for the corresponding AT. This may ultimately be necessary in an optimal coloring, but if it is not starting at the end may produce a lower cost feasible solution that prunes later attempts to look at higher cost solutions starting at the beginning of the IA.

```
int ColorBackward(IA γ) {
    if (γ.next) ColorBackward(γ.next)
    return (γ.nMinCost = Color (γ,0))
}

int Color(IA γ, int nAboveCost) {
    int nMinCost = MaxInt
    ∀ c∈ all_used_colors ∪ one_new_color {
        color γ and its RAs with color c
        nCutsMe = cost of other IAs cutting this newly colored RA
        for (i= γ_r -1; i≥γ_ac; i--) {
            place an invalidate, INV,  at i with color c
            nICut = cost of INV cutting other colored RAs
            nBranchCost = nAboveCost + nICut + nCutsMe
            if (γ.next) {
                if (nBranchCost  + γ.next.nMinCost < nBestSoFar)
                    nBranchCost = Color(γ.next,nBranchCost)
                else
                    nBranchCost = dummy value
            }
            else {
                if (nBranchCost < nBestSoFar)
                    remember new best across all nodes
            }
            nMinCost = min (nMinCost,nBranchCost)
        }
    }
    return (nMinCost)
}

final graph cost = ColorBackward (first γ)
```

Figure 8.27: IAColor

Step 4 says that he first node is always red, and the second node is either always red or green, and never blue. It is just a trivial symmetry observation.

Step 5 is why colors are searched before invalidate position is searched. Once the color of this invalidate is determined, the of previously colored IAs cutting this one's associated RA can be computed as nCutsMe. This is constant no matter the position of the invalidate. The different invalidate positions are tried, each one cutting some other RAs and that cost being summed to nICut.

The results of these improvements is shown in figure 8.27.

The near linear topology of ATs is not being exploited here as well as it should. Consider a given epoch, e_c, as a cutting point. If all IAs before e_c are colored, a snapshot of everything needed to finish the coloring is:

- coloring of nodes after e_c
- Invalidates after e_c
- RAs which cross e_c

If two snapshots are the same except for a permutation of the colors, they must have the same effect. For graphs with higher connectivity, trying to recognize this circumstance could be just as expensive as proceeding with the coloring. But for AT graphs with limited connectivity, it would be useful. We have not explored this because of the difficulty of implementing it and because the epoch graphs of programs we have actually examined have all been 'fat' because of their short length.

8.5.3 Loops

Loops are handled in the algorithm by unrolling them (§8.3.3) before analysis and then explicitly recognizing IAs which are repeats in the pattern. The *pattern driver* is the first *pattern duplicate* IA which is encountered. It has its invalidate location fixed by its *pattern master*. But, its color is still allowed to vary. Whatever color is used then determines the *cycle_offset* (definition 8.10) by reference to the *pattern master*. All other *pattern duplicates* now have their coloring and invalidation completely forced and the total cost for this trial can be computed.

As a final step, the *cycle_offset* is normalized to 1 to simplify the hardware implementation (§8.7). The color of the *pattern driver*'s *pattern master* is set to 0 and the color of the *pattern driver* is set to 1. Colors for other ATs are swapped with these as necessary to preserve the coloring pattern. Next, other ATs which need to have their colors cycled are assigned subsequent colors. The *cycle_size* is now the total number of colors involved in the cycling and all colors used for the color cycling ATs have colors < *cycle_size*. Other ATs preserve their color which is ≥ *cycle_size* throughout the whole loop. The actual cycling now simply needs to increment by one and reset when it exceeds the bound. This may have the incidental effect of lengthening the meta-cycle, how long before the pattern driver returns to its original color.

8.6 Colors, Ranges, & Regions

The simple model for TS' is that every section in every epoch is in some region of memory and directly associated with some color. To handle real programs this needs to be expanded to three distinct concepts:

- color — the actual color of the cache line to which invalidation is applied
- region — a compiler identified part of memory which needs to be given a color
- range — a run-time recognizable region which has a known color for a given epoch

Conceptually, compiler analysis identifies references to sections (regions) and assigns colors which will change by epoch. The run-time implementation then maps regions to ranges which contain the actual color, using either address bits, an explicit static approach, or a hardware detected range, an implicit, dynamic approach.

There are several aspects of this mapping which must be addressed:

1. A Static reference is not a dynamic one, i.e. a single source code reference may be in multiple regions at run-time.
2. Conversely, a single region may be involved in different static references and have a different color in each one.
3. The color of a reference changes on subsequent (outer) loop iterations.

To understand problem (1) consider the kernel of a relaxation computation:

```
PDO I=2,N-1
   DO J=2,N-1
      A(I,J)=C*A(I,J)+A(I-1,J)+A(I+1,J)
               +A(I,J-1)+A(I,J+1)
```

The interior of `A` is being written. The border is only read. However, `A(I-1,J)` will sometimes refer to the interior and sometimes refer to the border. Thus, coding a single color as part of the reference cannot preserve full reuse. In this particular case, loop peeling could solve the problem, but that could cause considerable code growth, could interfere with other memory hierarchy optimizations, and will, in some cases, not work regardless.

Such regions as the border in this example need to be recognized at run-time. We propose special hardware be used to recognize ranges of addresses, dynamically. Its function resembles that of the TLB. We refer to its implementation as the RLB.

In the previous example, there are two regions, interior and border, but there are five ranges:

```
A(2:N-1,2:N-1)
A(1:N,1)
A(1:N,N)
A(1,2:N-1)
A(N,2:N-1)
```

Each of the border ranges will have the same color at any given time, but they may all change on subsequent, outer, sequential loop iterations. Note that the number of regions and ranges can be both be greater than the actual number of colors for dynamic coloring.

This example immediately brings up two problems with the RLB mechanism: the number of regions can be much greater than the number of colors and simple regions from the analysis point of view might not be contiguous regions in actual memory layout (e.g. `A(N,2:N-1)` in a column major language). The latter problem must be directly addressed by having a relatively complex RLB that can recognize arbitrary vector sections (§8.7.2).

The problem of a large number of ranges is handled by using address (or instruction) bits to statically choose a region. The question of range becomes moot. Region then becomes identified with color (there are the same number of each), but the cycle mechanism still adds a level of indirection to this mapping (§8.3.3). This allows static coloring to work for a loop. For instance,

```
DO I
   PDO J
      A(J)+=
```

could produce the following static coloring:

```
MapRegionToColor(1,red)      — region 1 is red to start with
SetMaxColors(2)              — red & green
DO I
   CycleColor(1)             — increment meaning of all regions & INVs by 1
   INV 1                     — Invalidate red (region 1 color)
   PDO J
      A(J)+=                 — always in region 1 (initially red)
```

Here the color which region 1 represents switches back and forth between red and green on every iteration of the I loop. The meaning of the invalidate keeps switching too. This preserves all reuse. To see why, consider what the unrolled loop looks like:

```
PDO J
   A(J) +=                   — region 1 is now red
INV green
PDO J
   A(J) +=                   — region 1 is now green
INV red
PDO J
   A(J) +=                   — region 1 is now red
INV green
PDO J
   A(J) +=                   — region 1 is now green
INV red
...
```

This is the desired cyclic coloring for such a loop (§8.3.3).

Problem 2 is just the converse of problem 1. When references to a region are spread across several static references, coding the region implicitly as an address may require frequent changing. Consider the computation on the interior (ignoring the border for now) of the array for a 2-D heat flow (figure 8.11). In the first iteration, the read of `B` is red and the write of `B`, still in the first iteration, is green. A static coloring plus cycling at `I` loop iteration boundaries handles this with no fuss. An implicit coloring would need to re-map colors between the first and second epochs. Moving the cycle mechanism to cycle colors from the RLB and static colors would *not* address this problem. It would be possible to duplicate the color cycling mechanism for each RLB and increment the color in the middle of each iteration (in this case).

The static mechanism also has the advantage of being useful in some situations where the exact shape of the regions cannot be analyzed at all or cannot be represented by the RLB mechanism but there are distinct references generating different ATs. There are several other trade-offs involved. For instance, by moving the declaration of ranges inside loops they can end up applying to each reference independently and achieve any effect desired. Therefore, static region assignment has no theoretical advantage. However, the mapping cost can easily become an unacceptable overhead in such cases. Moving it part way can be useful though.

Almost anything that can be recognized as a region can also be handled by loop peeling at the cost of code growth. This will become useful when the number of hardware provided RLBs is exhausted. It also makes for an implementation of TS' that does not require an RLB. The codes we have examined show both mechanisms, dynamic RLBs and static color coding, to be applicable in every case. If this is usually true in general, there is simple trade-off between code growth and the RLB implementation cost, with hit rate being preserved in either case. As just discussed, there will still be cases where either method must sacrifice something. They might not be as common as one might suppose because in both cases regions are taken relative to the best quality of compiler analysis. A problematic program for the static approach is then one for which the compiler can recognize distinct sections but cannot restructure the code to separate the references that make up the section. There remain many special cases of sections which cannot be recognized by the compiler. Since their existence is never exposed to TS’, it does not matter that the hardware implementation cannot handle them.

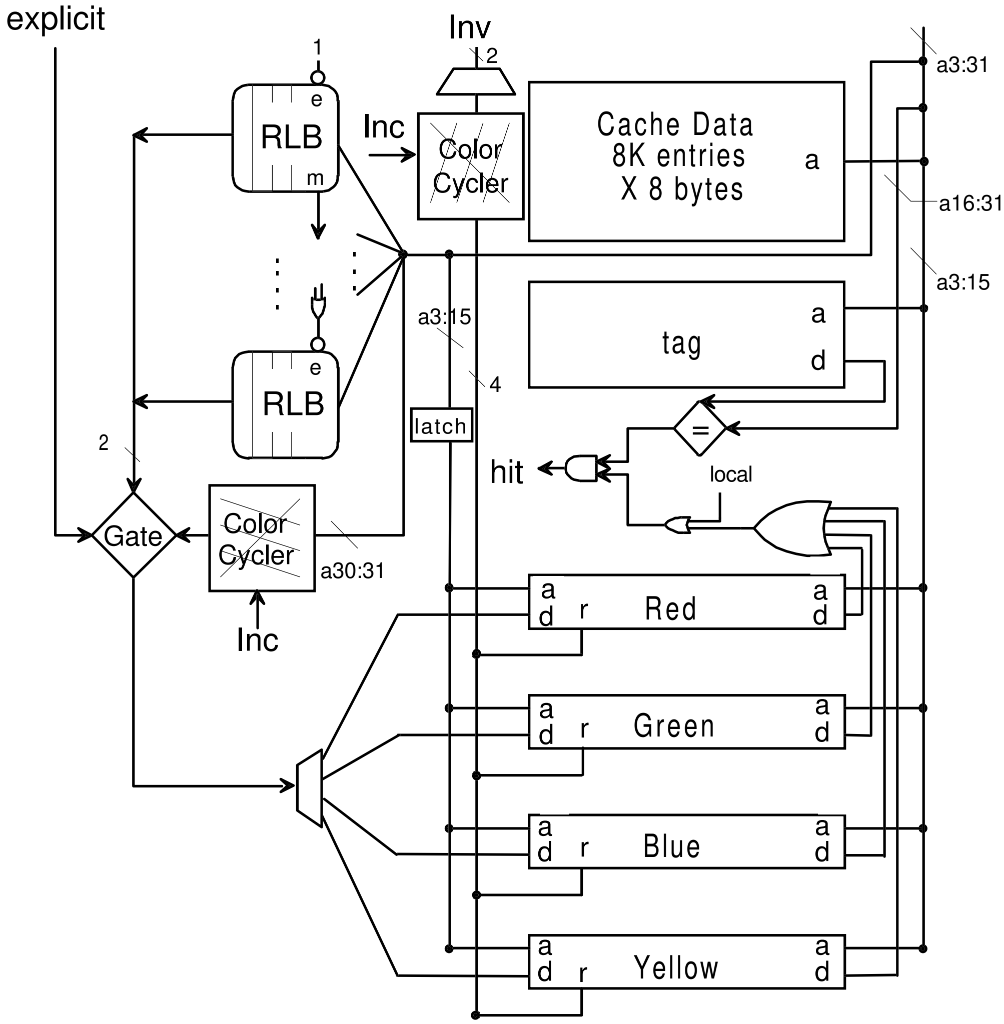


Figure 8.28: TS' Cache Hardware

8.7 Hardware Implementation

There are six essential parts to the hardware implementation of TS' (figure 8.28):

- Color bits
- Invalidate
- Logic to recognize non-zero color bits to confirm a hit
- The color cycling mechanism
- The RLB
- Timing, control, and RLB initialization logic (not shown)

The color chips are 1-bit wide with as many bits as there are cache entries (8K in the example). They are dual ported and have a chip clear signal. There is no *a priori* reason to use a particular number, but 4 seems to a be good number in practice. It also effectively utilizes two bits of address space for selecting the color.

Recognizing a hit requires only an OR of the output from all of the color chips to see if the current cache line has any valid color (assuming there is also a tag match). If the access is local or read-mostly shared data, the color check is simply bypassed with the *local* signal. All of this can happen in parallel with the tag match which takes at least as much time.

The input for which chip to set is simply a 2-bit number that is decoded to select a particular chip. This is the data value and not the enable signal. All color chips get a value, one of them gets set and the rest reset. The enable logic is not shown.

Choosing the color is a more complex problem. First, the color can be specified either implicitly or explicitly (TS [50] requires extra bits in the instruction or address be used to select a clock, we use a comparable number of bits here to select a color). The *explicit* signal chooses which to respect. Where it comes from is not specified as there are several different choices, all of which are equally useful in so far as TS' is concerned. It could be based on the address, it could use a bit of the address, or it could be a failure to match in any of the RLBs.

Second, the color cycling mechanism is needed for statically coloring references in loops (§8.3.3). It is a complete map that can transform any color to any other color. The most common case of changing the map is simply incrementing each input color to implement a cyclic coloring (definition 8.10). *cycle_size* is number of colors needed for the loop cyclic coloring and not the maximum number available in the system. There are cases where it is useful to have some references keep the same color throughout a loop while others are cycled. Since the color choice is arbitrary, the lower colors are always those to be cycled. This case is handled by a special instruction and increment signal that does this quickly (an increment of 1 is always sufficient, §8.3.3).

Some nested loops may require two levels of cycling. The inner loop can use the above hardware method. The outer loop then simply needs to completely reprogram the cyclic color map to achieve the same effect. The overhead cost of the complete map change should be insubstantial in comparison to the time to execute a complete inner sequential loop around a parallel loop.

The invalidate needs only to decode its color number and reset all entries for the selected chip. This value needs to be cycled in the same fashion as static colors. Depending on which is cheaper, this can be a unique cycle mechanism or the same one as is used to set the color (plus some extra switching logic).

8.7.1 TS' Invalidate

The invalidate needs to operate in O(1) time. In this, we follow in the assumptions made by previous local strategies which have also required such a mechanism. FSI [15] clears the “change bit” (epoch bit) at the end of every epoch by resetting all of bits on a chip. TS [50] calls the same functionality a “provisional bit” and also expects to clear it in fast O(1) time.

If such functionality is too expensive, TS’ can achieve the same effect by using one extra color. Instead of invalidating a color quickly at the end of epoch, that color chip is scanned sequentially during the subsequent epoch, invalidating each entry separately (this needs to be done in hardware, like refresh). If an epoch is long enough to generate as many addresses as there are cache lines, this will be adequate. For instance, our previous example has 8K entries which might be split into 4 chips all of which could be driven simultaneously requiring that an epoch last 2K cache cycles to completely cover the invalidate cost.

The required compiler change is a minor adjustment to IAColor (§8.5.2). Whenever a possible color is chosen for an IA, that color is marked prohibited for the epoch following the invalidate location. The set of available colors for subsequent IAs is narrowed based on the prohibited colors for the epoch their essential access appears in. This only requires one extra color because the epoch-long invalidate acts like one other variable being referenced which needs its reuse preserved for that period.

8.7.2 RLB — Range Look-aside Buffer (RLB)

The RLB is more problematic. In concept, it works like a TLB. There is one entry for each possible range. While the tag is being compared, all ranges are simultaneously searched. Which, if any, registers an in-range hit can then be used to select the proper color. While the RLB is moderately complex, there is only some small finite number of them implemented once per cache as special case logic.

The first problem with the RLB is that it will be slower than tag comparison . The tag comparison needs only to check for equality. The RLB, even in the simplest of scenarios, an inclusive range of memory addresses, requires a full comparison returning order and not just equality. Also, the tag match compares only the number of the bits in the tag. The RLB must compare the full cache line address. In the example (figure 8.28), the tag match needs to compare only the top 16 bits out of 32 (the others have selected the cache entry). The RLB must compare 29 bits (the full address minus the number of words in a cache line).

Instead of insisting that the cache cycle time be slowed sufficiently to accommodate this, we propose that the results of the RLB match be delayed until the next cache cycle. The data can still be returned immediately. The address is latched and used to select a bit in the color chip when the RLB result is ready. This does *not* affect coherence logic. The reason is that updated coherence information is not relevant until the next epoch, which is surely more than one cache cycle away (synchronization surely takes at least one cycle!). For the case where a cache line is referenced and then referenced again in the next cycle (before the color is set from the RLB), it will find a default coloring which indicates that it is *fresh*. This can be set while the tag is being written during a miss. It will be updated to the proper color from the RLB before any erroneous access can occur.

This is a problem only when two references in immediately subsequent cache cycles both reference shared data that needs coloring. Thus if the next cache cycle is not needed or references private data, there is no problem. When there is a conflict the current access can simply be stalled. If this is not adequate, then the color chips must be dual-ported (commercial processors are already dual-porting cache control bits, e.g. the PowerPC 604 [53]). The tag and data part of the cache need not be dual-ported for this purpose. Alternatively, the color updates can be stalled and buffered (since more than one could then be outstanding) and allowed to complete before synchronization completes.

The one cycle delay can be put to good use. The extra time can be used for the RLB to do an integer divide (the above mechanism could be extended to a two-cycle delay if needed to allow an integer divide to complete). With a single divide, vector sections can be recognized. Changing a zero test to a comparison allows 2-D sections where one dimension is contiguous in memory to also be recognized. Our tests show this to have substantial practical benefit. An RLB contains the following fields:

- *low* — low address of a contiguous range and of entire region
- *high* — high address of a contiguous range starting with *low*
- *step* — increment to next contiguous range
- *top* — highest address of region
- *color* — color for this region (subject to later cyclic mapping)

The RLB returns the given color on a match, which is found as follows:

- RLB match: (address-*low*) % *step* ≤ *high* & *low*≤address≤*top*

The RLBs are also prioritized. When an earlier one matches the later ones are disabled. This is accomplished simply by OR'ing the match signal of all previous RLBs and using it as a chip enable (figure 8.28). The only delay involved in this is an extra AND gate to check the enable on the simultaneously calculated output values.

Blocked LU decomposition provides an example of why these features are useful. A block of columns starting below the current pivot are all reduced in a single region (figure 8.29). In so far as the pivot columns are concerned, the ranges could be declared as follows (array references are understood to be addresses and not values):

- `A(1:N,1:N)` — the whole array
- `kb` — size of block (width of pivot rows and columns)
- `A(kk,kk)` — upper left corner of pivot rows

RLB #	low	high	step	top
1	`A(kk+kb,kk)`	`A(N,kk)`	`A(1,2)-A(1,1)`	`A(N,kk+kb-1)`
2	`A(1,1)`	`A(N,N)`	n/a	n/a

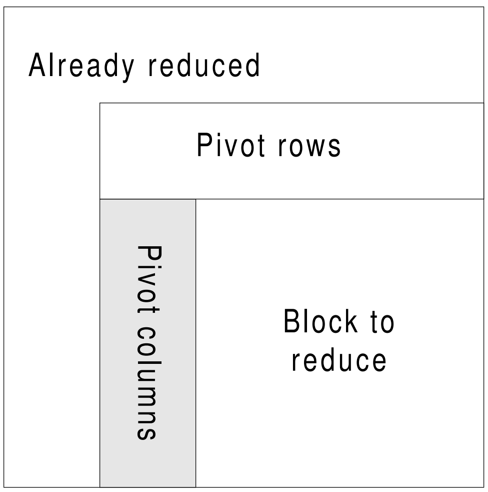


Figure 8.29: Column Reduction on LU Decomposition

The RLB will work when all of the following are true, applied only to subscripts that vary *inside* of an epoch, working from the slowest varying to most rapidly varying:

- Only the slowest *varying* subscript can have a non-unit step
- The second slowest *varying* subscript must refer to a contiguous range
- The remaining subscripts must refer to the full range

For instance, in a column major language, if `A(1:N,1:N/2,1:N:2)` and `A(1:N:2)` are the reference footprints inside of the `PDO`, they can be represented (regardless of how complex the total reference is including subscripts that vary outside of the `PDO`). However, `A(1:N:2,1:N:3)` could not directly be represented with a fixed number of ranges.

The override mechanism can also be used to pick up some of these cases. Mapping `A(1:N:2,1:N:3)` to region 1 could be handled by mapping `A(1:N,1:N)` to region 1 then mapping `A(2:N:2,1:N)`, `A(1:N,2:N:3)`, and `A(1:N,3:N:3)` to region 2 (the last two could be mapped with a single range since column 2 and 3 are contiguous in real storage). More likely this would be a candidate for an explicit coloring.

8.7.3 Color Addressing

Figure 8.28 is a possible implementation of the above ideas. This figure does not include the usual cache logic nor does it include any of the timing signals. Logic for loading the RLB is also omitted.

For the sake of example, the address is assumed to be 32 bits wide, there are four colors, a cache line holds 32 bytes (5 bits worth of address), and the cache is direct mapped. Consider first how a hit a is detected. The tag RAM must find a match at the right spot, as usual. In addition the same address applied to the tag RAM (bits 5 - 20 of the address) are applied to the color tag RAMs. Each of these is addressed by 14 of the 16 bits. The remaining top two bits are then used to select which of these four RAMs to enable. This is just shown for the 'yellow' RAM. It happens simultaneously for the other three colors. The result is then the OR for all four colors. If any of them are true and this is a shared access, then the hit is accepted as being to valid data.

While this is happening, the range comparator starts operating on the complete address (except the last 5 bits) and the last 16 bits are latched. Next cache cycle, the latched address is used to select one of the four cache tag RAMs, write enable it, and connect its data input to the range tester. The result of the range tester is now available and can be written back. This happens simultaneously with the current cache access. If they both need the same tag RAM, the equal comparator detects this and stalls the current access until the result of the range test is written. This happens simultaneously for all four colors. At most one of them will be true.

For explicit colors, the top two address bits are used to select the color. These are decoded to choose the right color. Like the implicit color, these are delayed until the next cycle so that a complete RLB miss can be used to signal the use of an explicit color.

8.7.4 Region to Range Mapping

IAColor (§8.5) produces its results as if every region in its analysis were a separate entity. These regions need to be mapped back to the real program as either implicit ranges or explicit reference colors. There are several ways to do this (§8.6). There are some additional constraints that must be considered:

- Some simple sections cannot be matched by any set of RLB entries.
- RLBs can easily be exhausted, especially for those regions needing two entries
- The prioritization of RLBs should be effectively utilized.
- Loop transformations (e.g. peeling & strip mining) can be expensive

We have no complete solution to this problem but the following heuristics, applied in order, have proven adequate for all cases we have examined:

1. Apply resources to inner loops first.
2. Use explicit colors where possible.
3. Use RLB entries that map directly to regions (§8.7.2) where possible.
4. If this subsumes any explicit colors, remove those colors.
5. Use RLB entries that map directly to regions, but for sub-regions which have already been assigned and given a higher priority RLB.
6. Use any remaining RLBs to approximate remaining sections as best possible.
7. If this is insufficient, expand already managed regions to cover unmanaged ones

Step 7 must ultimately be sufficient because the coloring originally produced by IAColor was guaranteed to be correct for the number of colors available on the machine. The only thing that can go wrong in the mapping process is that some region takes too many RLB entries because of its shape.

8.7.5 TS' without an RLB (TS' versus TS)

The RLB is probably the most complex and expensive piece of hardware required by TS'. It is also the most problematic because it is tied to the quality of the compiler analysis (even if the compiler could recognize a section more complex than a vector section, the RLB could not utilize it). As previously noted (§8.6), the RLB is not strictly necessary. References can be explicitly colored and loop transformations used to resolve many, though not all, coloring conflicts. The unresolved conflicts compromise the miss rate; they do not affect correctness.

In such a configuration, TS' has hardware requirements similar to those of TS. The "provisional bit" of TS is duplicated to become the color bits of TS'. The color bits also replace the time stamp bits so the total storage is similar, but each of the colors requires the ability to be cleared whereas only the one marking bits in TS needs the ability to be cleared. The extra instruction word or address bits needed by TS to select the clock become the explicit color selection of TS'. Since TS is "lazy" (§8.2.2), it needs enough bits in the clock identifier to enumerate every array in a program. The original paper did not suggest a number, but it is surely 2 bits (4 arrays in a program) and probably 3 or 4 bits. The remaining logic to detect a hit and set the marking bits is distinct in the two cases but comparable in complexity. Whenever a clock in TS would be incremented such that an array becomes stale, TS' can invalidate the color corresponding to that array. That color then becomes available for reuse on a possibly different section. TS, by its nature, cannot reuse a clock identifier. Thus, for the same number of selection bits (clock or color) , TS' is strictly an improvement over TS. Because TS' can, over the full length of a program, refer to more regions than TS, it opens the door to using section analysis to good advantage in way that would be difficult for TS.

8.8 Multi-word Cache Lines

8.8.1 Introduction

The essential logic of all local strategies mentioned is that cache lines are one word. However, real machines use multi-word cache lines to achieve higher hit rates due to spatial locality and to a lesser extent to save on cache hardware. Since hit rates are the focus of coherence strategies, the reuse gained via spatial locality must be preserved. First, some definitions:

> Definition 8.14: Load – a word which is loaded into cache, but not (immediately) accessed, because of another word in the cache line being accessed.
>
> Definition 8.15: Tap – a different word in the cache line is accessed and hits. The non-accessed words are said to be *tapped*.

Simply applying TS' (or any local strategy) to the whole cache line will result in either an overly conservative program or incorrect execution. The reason is that false sharing can result for words which are *loaded*. The local schedule theorem (3.5) no longer directly applies because race conditions are present. Global strategies avoid the problem by invalidating (or updating) the stale word atomically, before it can be accessed incorrectly.

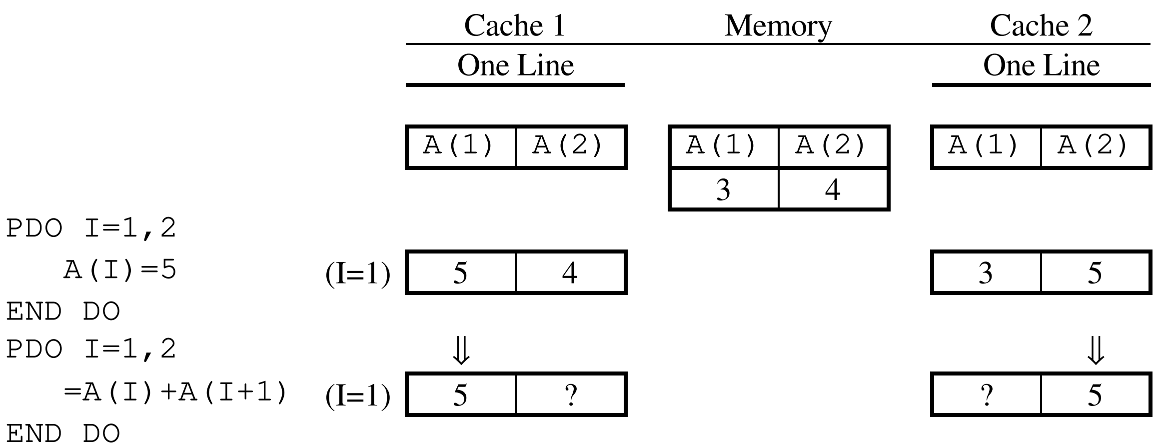


Figure 8.29: Stale Access on Multi-Word Cache Lines

To understand the problem facing local strategies, consider figure 8.29. Cache lines are two words long. `A(1)` and `A(2)` are in the same cache line. The schedule is:

e_1:$\mathtt{A}_w$`(1)`[1], $\mathtt{A}_l$`(2)`[1]
e_1:$\mathtt{A}_l$`(1)`[2], $\mathtt{A}_w$`(2)`[2]

Both $\mathtt{A}_l$'s are stale the moment they are loaded. Within e_1 this is not a problem since the `PDO` guarantees no stale access will occur. Before e_2 begins the strategy must invalidate $\mathtt{A}_l$`(2)`[1] and $\mathtt{A}_l$`(1)`[2].

If nothing special is done to a local strategy, the control bits will be understood to apply to the whole cache like, just like the tag does. In that case, the invalidation of $\mathtt{A}_l$`(2)`[1] will necessarily invalidate $\mathtt{A}_l$`(1)`[1] also since they share the same control bits, even though the latter need not be invalidated. This is summarized without formal proof as:

> Observation 8.16: Necessary Logic for Multi-Word Cache Lines: To preserve inter-epoch reuse in a local strategy with multi-word cache lines, at least one marking bit per word is required.

In this trivial piece of code (figure 8.29), the problem can be addressed by strip mining the loop so that false sharing is completely avoided. Where possible, one wants to do this. Also, data will sometimes be read-mostly, i.e. written once before any epoch in which it is read on a different processor and then only read thereafter. In such a case, coherence can be ignored. Even where there are multiple writes, but many more read epochs in between, this can be applied by invalidating everything of concern before the writes. There are many programs for which these approaches will not work, e.g. because of alignment conflicts between loops or unanalyzable subscripts. For the sake of explanation assume figure 8.29 is for some other reason complicated in such a fashion as to prevent these approaches from being used.

There is also the problem of false sharing between different variables or between different columns of the same array (for column major languages). In most cases, this can be addressed by padding in the allocation to fill out the cache line. The storage requirements are reasonable. The sharing patterns of scalars are such that they can be handled without being put in separate cache lines. For arrays, it wastes on average one half of a cache line for each column. This is an insignificant amount of storage. For those cases where this is not possible (e.g. equivalencing), the analysis can be made more pessimistic to still achieve correct execution. False sharing between columns is then treated like false sharing within columns.

If padding can be successfully applied, intra-column false sharing is only a problem when the parallel index variable indexes the most rapidly varying subscript. In other cases, padding can solve the false sharing problem without further effort.

However, to preserve coherence when the compiler cannot *a priori* avoid false sharing, it is necessary to have control bits per cache word, except under the most unusual circumstances. In other words, the logical cache line must stay one word for local strategies to work. However, spatial locality can still be utilized by bringing in a whole physical cache line at one time. One obvious approach is just to duplicate all of the control bits for each word. The tag need not be duplicated, but the valid bit, time-stamp, epoch bit, etc. must all be duplicated.

A TS1 style strategy can often be applied to reduce this number to two bits per word. Every word just needs to code which state it is in, *fresh*, *valid*, or *stale* (figure 3.4). In many strategies this is relative to the full set of control bits which still apply to the whole cache line and will not take the explicit form as found in TS1. For instance, it can be applied to a TS as a 2-bit offset value from the actual time-stamp [16]. The actual invalidation still occurs on the total cache line bits and thus the overhead of TS1 is not a problem. It seems that two bits are usually sufficient.

		Cache 1: One Line			Memory		
	I	A(1)	A(2)	A(3)	A(1)	A(2)	A(3)
					3	4	5
`PDO I=`							
`   =A(I)`	1	3	4	5			
`END DO`		*fresh*	*valid*	*valid*			
`PDO I=1,N,2`		*valid*	*valid*	*valid*			
`   A(I+1)=A(I+2)`	1	3	5	5	3	5	5
`END DO`		*valid*	*fresh*	*fresh*			
`PDO I=`		*stale*	*valid*	*valid*			
`   A(`*f*`(I))=A(I+2)`	1	3	5	5			
`END DO`		*stale*	*valid*	*fresh*			

Figure 8.30: Three States Are Essential

Figure 8.30 shows that there are cases when three states are simultaneously needed in one cache line. After A_r(3) in e_3, the situation is:

A(1) is stale because of e_2:A_w(2)
A(2) is valid because it was accessed in the previous epoch
A(3) is fresh because it was just accessed

Any attempt to squeeze this into two states (and one bit) would result in extra misses. The choices are:

- A(1) could be made valid by unnecessarily missing on A_r(3).
- A(2) could be downgraded to stale, possibly causing an unnecessary miss in e_3 (at some later point not shown).
- A(3) could be downgraded to valid, possibly causing an unnecessary miss in e_4.

It would then appear that in general two bits are also necessary. Tightening this to the three states actually required, we summarize this as follows:

Observation 8.17: Three States Necessary.

For an n-word cache line, $\lceil n \log_2(3) \rceil$ bits are both necessary and sufficient to record the state of each word in a local coherence strategy. In practice this may simply be $2n$. This is in addition to recording the cache line state.

8.8.2 Pseudo Race Conditions with Multi-Word Cache Lines

The loads that occur with multi-word cache lines often represent race conditions. We refer to this situation a pseudo race condition since it exists only for loads and the result of the computation cannot be affected. Such race conditions are tolerable since they will not be accessed before the next synchronization (otherwise they would be some kind of access). Global strategies can take advantage of this in a way that local strategies cannot.

Consider figure 8.31. Each processor writes half the cache line. An ideal local strategy will need to invalidate `A[1:2]` at the end of the first epoch. Since, $\texttt{A}_{ac}(1)[1]$, it can remain in the cache, switching from fresh to valid. However, `A(2)` on p_1 will need (correctly) to be invalidated despite being loaded. The situation is symmetric is p_2. In e_2, a miss will occur for `A(2)` on p_1 and for `A(1)` on p_2.

A global strategy can take advantage of the fact the one update actually does occur before the other one. For example, if the write on p_1 happens before the write on p_2. Then when p_2 misses, it will load the already updated version of `A(1)`. That version can then hit in e_2. e_2:`A(2)`[1] will still need to miss though. Even in a release consistent model, this effect can still occur because the change may propagate early, even though it is

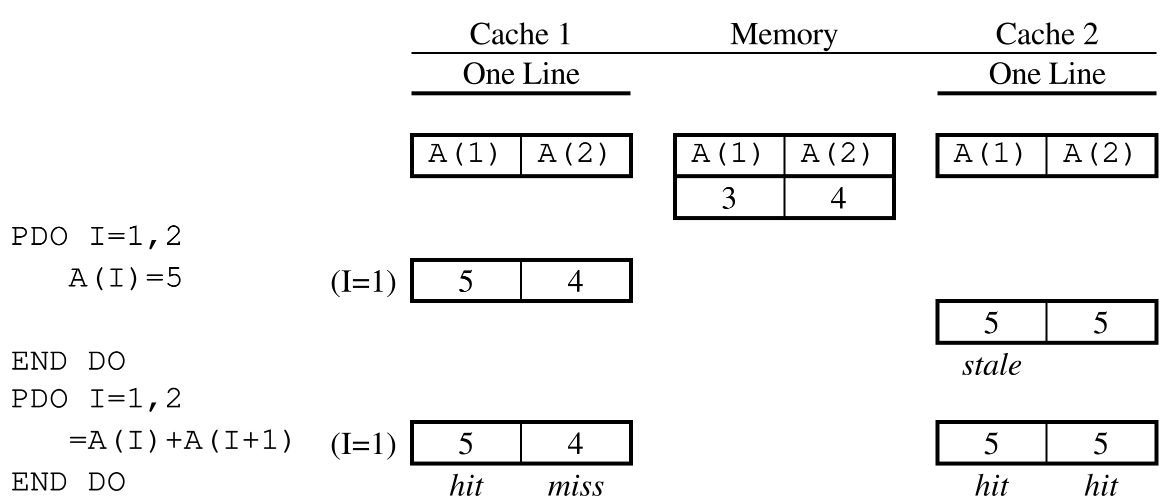


Figure 8.31: Global Strategy Race Condition

Area:	Tag Area		Word		Word		Word		Word	
Item:	Tag	CM	C	Data	C	Data	C	Data	C	Data
Bits:	15	4	2	16	2	16	2	16	2	16

Example Cache Line Layout for TS' with Multi-Word Cache Lines

	CM:	Color mask, bit set for each valid (or fresh) color
	C:	Color of a given word, binary index to CM bit
Variables:	VC:	Valid color for this reference (from range match), binary
	FC:	Fresh color for this reference, binary
	RCM:	Range Color Mask = $2^{vc} \mid 2^{fc}$, unary

Condition:	Hit:	2^{c} & CM != 0

	Miss:	CM = RCM; C=FC
	Load:	C=VC
Actions:	Tap:	if (2^{c} & CM ==0) C=(C' s.t. $2^{c'}$ & RCM == 0)
	Hit: Inv:	CM |= RCM; C=FC (after Tapping) $\forall$ cache lines, CM &= $\neg 2^{\text{inv color}}$

Figure 8.32: TS' Multi-Word Cache Line Algorithm

not promised to. This effect appears to be minimal. We present data in chapter 9.

This does not destroy the *idealness* of an ideal local strategy because that is defined in terms of programs with data races (§7.5). The above situation represents utilizing a race to good advantage.

8.8.3 TS' Multi-Word Cache Lines

TS' does not explicitly recognize *fresh* or *valid* states. Instead they are implicit in particular colors. This suggests an implementation: use the 2 bits per word (observation 8.17) to index which of the four colors was *fresh* when this word was last accessed or *valid* when it was loaded. This hinges on there being only four colors. If there were more, this strategy would not be a good one. Instead a more complex encoding that still uses only two bits would be needed.

Figure 8.32 details this approach. For the unit cache line case, the color mask in each cache line had at most one bit set because the single word could only have one color. For the multi-word cache line case, the color mask can have any number of bits set, one for each *valid* color (one of which is also the *fresh* color). The *C* bits in each word are then a binary index to which color was appropriate for this word when it was loaded or accessed. That it is encoded in binary is not a problem because it is only checked when the word is actually referenced and not for any whole cache operations. A two-bit decode once per cache controller is not a significant amount of overhead. The range match mechanism must now return two colors instead of one. The color it returned in the unit cache line case is now the fresh color, FC. The valid color, VC is the color to use for loads (it will generally be the most recent previous fresh color used for this line). A hit is then simply the indicated bit in the color mask being on. It doesn't matter whether it corresponds to valid or fresh.

A miss sets the CM to have exactly two bits on, the current *fresh* and *valid* colors. The C bits for each word are set to one of those two as appropriate. However, for a hit it is important to keep all current valid colors, while adding some possibly new ones. The new *fresh* and *valid* colors are OR'd into the CM. Once a cache word is loaded (or accessed) it stays valid until its particular color is invalidated, regardless of how other references affect the cache line.

The invalidate is identical to the unit cache line case. It resets a particular chip which clears a given color for all cache lines. Any word which indexes the color in question will be found stale on its next access. No other action operates on the whole cache at once. There is no per cache line logic needed except for the chip level reset.

There is one twist to this: the color a word, *x*, refers to may be invalidated then set again (by another word, *y*, in the same cache line) before *x* is accessed again. In such a case, *x* appears valid but is actually stale. This can be handled by using a *tap* to detect such situations. Whenever the CM is about to be changed, any C's which index invalid colors must be changed to some other currently invalid color. Thus, once a word's color becomes invalid, that word stays invalid, regardless of how the color is reused.

Figure 8.33 is an example of the multi-word TS' algorithm. Four word cache lines are used. In e_1, `A(1)` is referenced. The current (fresh) color is 0 yellow and valid is 1. `A(1)`'s C index is set to 0, meaning that it is associated with color 0. `A(2)` and `A(3)` are merely loaded and therefore have their color set to 1. The color mask for the line is

Epoch	Pid 1 Accesses	Effect	Range		Cache Line 1			
						A(1)	A(2)	A(3)
			FC	VC	CM	C	C	C
1	A_r(1)	miss	00	10	0101	00	10	10
2	A_w(1)	hit	01	00	0111	01	10	10
2	A_w(2)	hit	01	00	0111	01	01	10
	INV 0101				0010			
3	A_r(1)	hit	10	01	0110	10	01	00
3	A_r(3)	miss	10	01	0110	10	01	10

Figure 8.33: TS' Multi-Word Cache Line Example

0101 indicating that either a reference to color 0 or 1 would be a hit. In e_2, e_1's fresh color , 0, becomes valid. The new fresh color is 1. A(1) hits because color 1 in CM is still valid (regardless of the RCM). After the hit, CM is OR'd with new RCM, 0011, leaving it 0111 indicating that colors 0, 1, & 2 are all still valid.

This change makes green words invalid and blue words valid. However, merely accessing A(1) does not change their status. Therefore, *C*'s in other words are updated. In this case, A(2) & A(3)'s C indices are changed to yellow, the new valid color.

At the end of e_2, color 0 is invalidated because it was the fresh color in e_1 and accesses such as A_r(1) will have become invalid by virtue of A_w(1) in e_2. The invalidate proceeds by clearing the color 0 bit in the CM for every cache line. Similarly, color 2 which was valid in e_1 has not been invalidated and has now become stale (it could have been invalidated after e_1) .

In e_3, color 0 is *fresh* again. Even though *fresh* and *valid* have merely swapped colors since e_2, no confusion or captures of state occur. A(1) hits like it did in e_2. Its C index points to color 1 which is still set in the CM. A(2)'s C index is 1 which is valid and left unchanged. However, A(3)'s C index is 2 which is not set in the CM since A(3) is invalidate. To avoid having it captured by color 2 becoming the new fresh color, it is *tapped* and has its color changed to 0 so that it is still invalid. Next, A(3) misses because its C index points to invalidate color. Its C is then reset to the fresh color. However, nothing happens to the C index for A(2) and A(1). Since they already point to the *fresh* color, they do not have their C index set to the *valid* color.

8.8.4 TS' Analysis for Multi-Word Cache Lines

The coloring algorithm needs to return a valid color and a fresh color in the multi-word cache line case, instead of merely the fresh color as in the unit cache line case. The valid color is the most recent previous color used for this line. This has several implications for the coloring algorithm.

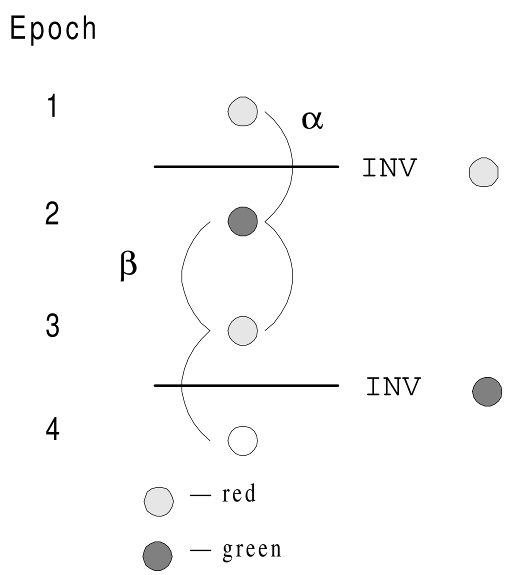
When looking at an essential write for which there is a single essential access, the valid color is merely the color used for the access. When there are multiple essential accesses, the color of any of them could be used for loaded words. All must be acceptable since coherence will be preserved no matter which essential access is still in cache when then essential write occurs. For optimal colorings, the color of all essential accesses will be the same (theorem). In other situations, the most recent (as determined statically) essential access should be used. Its color is guaranteed to be preserved longer than that of any other essential access *after* the essential write has already been reached, which is the case for a load, (theorem).

Essential accesses and essential reads will often be essential writes of a different AT. In that case, they should be handled as an essential write (e.g., in figure 8.34, both colors of β_{ac} are determined by the color of α_{w}). If they are not also an essential write, then either there is no write in this epoch for this variable or this is an apparently initializing write (if there were prior accesses, this write would be the essential write of some AT). If there is no write, the valid color can be set to be the same as the fresh color. This handles most cases.

There is one case left which deserves special attention: a write for which there is no valid color available. This can happen for two reasons. It can be an initializing write (there are no previous accesses) or it can be a write for which all RAs have been invalidated and there is no invalidate that crosses the IA. Figure 8.34 is an example of this. There are two important conditions here:

- α_{ra} reuse must be lost because other ATs (not shown) force this situation
- No invalidates of any color can be tolerated in α_{ia} because of other reuse.

If condition one were not true, there would be some alternate coloring that would allow α_{ia} to invalidate red thus preserving reuse on α_{ra} and making red available as the *valid* color at α_{w}. This would be legal since it would be invalidated during α_{ia}. Condition two says there are other RAs of such priority that no invalidate, of any color, can be tolerated




```
PDO
   A(I)=...
ENDDO
PDO
   A(I)++
ENDDO
PDO
   A(I)++
ENDDO
PDO
   ...=A(I)
ENDDO
```

Figure 8.34: No *Valid* Color Available

during α_{ia}. If any color invalidate were present there, the *valid* color for α_w could be whatever is invalidated during α_{ia} regardless of the fact that α_r is red. This is a rare circumstance. We have never encountered any program that comes close to filling these conditions. None the less, it is possible.

Consider the options available at β_{ac} to understand the problem:

- Loads are green (*fresh*) — will incorrectly be valid at β_w since no invalidate can intervene.
- Loads are any invalid color — Spatial locality will be lost within β_{ac}.
- Loads are some arbitrary *valid* color — same problem as fresh, i.e. there are no colors available to use as *valid* as it should be used.

The solution to this is to utilize the fact that there are no *valid* colors available at β_{ac}. Thus, on entry to β_{ac} all cache lines are guaranteed to have every word *stale*. During the execution of β_{ac} any cache line which has a *fresh* word must have loaded all other words. This guarantees that if any word in a cache line is *fresh*, all other words are at least *valid*. Loads in β_{ac} are simply marked with a *stale* color. That produces the proper effect in all subsequent epochs. In this epoch, any stale access for which some other word in the cache is *fresh* is treated as if it were *valid*. This can also be used to handle the initializing write problem. This approach cannot be generalized because if any *valid* word can enter an epoch it would cause truly *stale* words to be falsely recognized as *valid*.

The hardware implementation is simply that the range cache return a signal that causes this special circumstance to be handled. That datum can be coded along with setting the range. It does not require any extra cache bits or instruction bits.

8.8.5 Overlapping Ranges

When one cache line has two different ranges (§8.6) present, there is a conflict for deciding which color mask to use. The problem also exists for static coloring. The static color in the address bits specifies the fresh and valid color for this word only. There is no guarantee that other loaded words should have the same valid color.

For dynamic coloring, the problem can be directly addressed by OR'ing the color masks for every range. This preserves correctness since colors are globally valid, e.g. if region 1 uses red to mean valid and region 2 uses green to mean valid, then red and green are both valid. The individual per word C indices then distinguish which color the individual words refer to based on the range they are in.

For static coloring, the problem is more difficult because it may be impossible to statically predict which regions other words in the same cache line should be in. All loaded words, regardless of their preferred region, can be safely presumed to have the same valid color as the valid color used for the region the accessed word is in. This must be correct because valid colors are globally valid. However, this estimation can compromise the hit rate. Consider figure 8.35. `A(1,I)` and `A(2,I)` are used differently and can be in different regions. However, `A(1,1)` and `A(2,1)` are in the same cache line. When `A(1,1)` is accessed, `A(2,1)` is loaded. The C index for `A(2,1)` should be set to the same as it would be in e_2 (since it is not written in e_1). However, there is no obvious way to know that `A(2,1)` is not the same region as `A(1,1)`. Thus, its C index is to set to be valid for the `A(1,1)` region. At the end of e_1, it is invalidated because it appears that another processor may have written to it (e.g. `A(1,2)` was written on p_2 and `A(2,1)` appears to be in the same region).

```
PDO I=
   A(1,I)=
ENDDO
PDO I
   A(2,I)=A(2,I)+A(1,I)
ENDDO
```

Epoch	Pid 1 Accesses	Effect	Range		Cache Line 1		
						`A(1)`	`A(2)`
			FC	VC	CM	C	C
1	A_r`(1,1)`	miss	0	1	11	01	-
1	A_l`(2,1)`	miss	0	1	11	-	00
	`INV 01`				10	01	00
2	A_r`(1,1)`	hit					
2	A_r`(2,1)`	miss					
2	A_w`(2,1)`	hit					

Figure 8.35: Range Overlap

In this case, two different static regions sharing the same cache line, TS' has a granularity of analysis equal to one cache line instead of one cache word (as TS1 retains). This does not keep it from still being *ideal* for that granularity.

8.8.6 One Bit per Word

Two bits per word is a significant overhead. It is worth examining what must be sacrificed to lower this cost. If no extra bits per word can be tolerated (only some fixed number for the whole cache line), coherence is essentially reduced back to a static strategy (observation 8.16).

One bit per word presents some interesting trade-offs. With one bit per word, there can be at most two different states active in a cache line at one time, but that does not restrict the total number of states available, i.e. the per-cache-line bits code a subset of two states out of which the per-word bits choose.

The first two cases are epochs for which there are no writes (to the array section in question) and epochs for which no there are no incoming valid colors (as just discussed in §8.8.4). In these cases there can only be two states and one bit per word is sufficient to preserve all information. Another important assumption is that strip mining has failed to remove false sharing (§8.8.1), i.e. compiler analysis thinks it is probable that $\mathtt{A}_w\mathtt{(1)}$[i] will occur despite $\mathtt{A}_{ac}\mathtt{(2)}$[j], j≠i.

If none of the special cases apply, there are words in three different states in the cache line. Choose f, v, and s to be some particular words which are *fresh*, *valid*, and *stale* respectively at the end of an epoch e_3. The possibilities for action can be restated:

1. Miss and load s
2. Make v stale
3. Make f valid

Any of these would reduce the number of states in the cache line to two. Choice (1) will not be a mistake if s will be accessed before the line is evicted. The compiler will not likely be able to detect this situation because it does not know the schedule next epoch and it already thinks another processor will write some word (possibly s) in the same cache line. Having some special knowledge that s will be accessed on this processor given f_{ac} seems unlikely. Also, the worst case penalty for this given line and circumstance, no matter which choice is taken, is one miss, therefore rushing to suffer the penalty cannot be of any benefit.

Choice (2) will be a mistake only if v is accessed before s and before it (v) is invalidated. If s_{ac} occurs first, that miss would bring a valid copy of v and its having been marked *stale* would not cause any harm. That is, all of the following events must occur (loosely specified):

e_1:v_{ac}[i] — it must previously have been brought into cache
e_1:¬s_{ac}[i] — this is not accessed and made *valid* next epoch
e_2:$\mathtt{A}_w$ — some other part of the same section must be written in this epoch
e_2:f_{ac}[i] — there is some *fresh* access that occurs first
e_{2+}:¬s_{ac}[i] — there is nothing else causing a miss before v_{ac}
e_{2+}:¬Inv — no invalidate clears the corresponding color
e_{2+}:v_{ac}[i] — the valid word is accessed

Choice (3) will be a mistake only if f is accessed in the next epoch and s is not accessed first. The event list:

e_1:v_{ac}[i] — something must previously have been brought into cache

e_1:$\neg s_{ac}$[i] — this is not accessed and made *valid* next epoch

e_2:$\mathtt{A}_w$ — some other part of the same section must be written in this epoch

e_2:f_{ac}[i] — the *fresh* access occurs

e_{2+}:$\neg s_{ac}$[i] — there is nothing else causing a miss before f_{ac}

$e_{>2:}$$\neg$Inv — no invalidate clears the corresponding color after f becomes *valid*

$e_{>2}$:f_{ac}[i] — the valid word is accessed

We have not tested either of these scenarios. It is an open question what percent of the hit rate any strategy would sacrifice to lower the overhead to one bit per word.

Chapter 9

Miss Rate Testing

We collected miss-rate data on a small suite of scientific Fortran programs for each of the four coherence strategies developed in this thesis (CTV, CTV+, TS1, and TS'), previous local coherence strategies (FSI, TS), an oracle global strategy (which exists only in principle), and a real implementation of a VM-based global strategy (K&S). This data examines the effect of different strategies as well as the impact of program size, number of processors, total cache size, cache line size, and cache associativity.

The latter parameters have little impact on the effectiveness of coherence strategies. The dependence pattern of the program is principal factor. Our data shows that for analyzable scientific computations, local strategies can have an optimal miss rate (all misses are logically necessary). This is the same as for global strategies. The data also shows that section analysis matters and that dynamic local strategies can be significantly better than static local strategies. Finally, the data shows that TS' can achieve an ideal miss rate in many circumstances and failing that, it still compares favorably to TS (the best competing local strategy) and K&S (a plausibly competing global strategy).

This rest of the chapter begins by discussing the simulator we use (§9.1.1). The simulator's tasking model is relevant to how programs are modified for testing. Next, we discuss how programs are analyzed and modified so as to produce the right statistics for the different coherence strategies (§9.1.2-9.1.4), and then the cache organization assumptions that are used for that testing (§9.2). After this background, we discuss the actual data as it applies to the different issues it resolves:

- Ideal local versus global strategies (§9.4.1)
- TS1, an ideal local, versus TS, the best previous local (§9.4.2)
- Static strategies versus dynamic ones (§9.4.3)
- TS', how it compares to other local strategies (§9.4.4)
- How irregular programs behave, especially with TS', (§9.4.5)
- How TS' compares to a VM-based global strategy, K&S (§9.4.6)
- How different static strategies compare to each other (§9.4.7)
- How changing cache parameters are largely irrelevant (§9.4.8)

We follow the analysis of the miss rates with a discussion of the implications for execution speed (§9.5). Finally, we summarize by putting the different strategies in relation to one another by looking at granularity of analysis versus miss rate (§9.6).

9.1 Testing Methodology

The principal steps used in testing the different coherence strategies were:

1. Section based dependence analysis, manually (§9.1.2)
2. Building the ATs and coloring the resulting graph, automatically (§9.1.3)
3. Converting the coloring to program instructions, manually (§9.1.4)
4. Running the program and measuring the miss rate, automatically via a simulator (§9.1.1)

All of these steps were needed for TS'. We discuss these steps in this section principally as they apply to TS'. Some of these steps are not applicable to testing other strategies.

For global strategies, it was simply a matter of modifying the simulator code to properly emulate the specified global hardware. Only step (4) was necessary. The programs were run without modification (except for some directives specifying shared versus local data). For previous local strategies, the same directives as used for TS' (§9.1.4) were manually added to produce the right effect. The use of explicit directives is not the way these other strategies were intended to be implemented, but the set of directives needed for TS' is rich enough to be able to produce exactly the same memory access pattern (and miss rate) for other strategies as would the originally intended implementation. This methodology is adequate for the previous strategies since we are concerned only with measuring their effects and are not interested in how to actually implement their compiler aspects. In addition to program directives, each of them required code in the simulator. This was written to properly simulate the suggested hardware implementation.

For CTV, CTV+, and TS1, standard dependence analysis was used (instead of section analysis). Invalidate calls were added as previously discussed in the chapters about these strategies (chapters 5, 6, 7). As with all other strategies, code was added to the simulator to implement the hardware part of these strategies.

9.1.1 RPPT

The final program is run under the RPPT[18] simulator to get miss rates. This simulator operates by modifying the assembly code (Sparc in this case) to trap every global memory reference which is then passed off to an architecture simulator. This allows for faster execution than a full instruction simulator, but the architecture simulator still has complete information about cache, memory, and program state. The program still runs through essentially the same set of instructions as if it were really running. The final result reflects how a real implementation would behave with respect to memory access patterns. Memory instructions complete to real memory just as they would in the absence of the simulator.

The parallelism model is multiple threads all executing the same code. Non-root processors jump over serial code and wait at a barrier for the root processor to finish. Processors share work in parallel loops based on some function of their process ID and then rendezvous at a barrier before proceeding to the next serial section. RPPT slices between the tasks in a fine grain fashion. All of the difficulties of parallel programming are exposed except that memory operations are guaranteed to be atomic. As part of the modification to handle local coherence, stale accesses are detected and reported so the single real memory does not cover up any bugs in the coherence algorithm.

9.1.2 Dependence Analysis

TS' requires section analysis and not merely dependence analysis. There are many systems which perform accurate subscript dependence analysis [29] but there were none available to us that presented this information as sections. Many compiler transformations (e.g. SPMD layout) require a similar analysis. We expect it to be available in future systems. To understand the difference, consider figure 9.1.

```
PDO I=1,5,2
   =A(I)
ENDDO
PDO I=1,N
   A(2*I)++
ENDDO
PDO I=5,N
   =A(I)
ENDDO
```

Figure 9.1: Section Analysis Example

Dependence analysis would correctly find a dependence between the first and third epochs based just upon `A(5)`, but TS' needs that information explicitly for both references and not merely the fact that dependence exists between them. It also needs to split individual references in multiple sections based on the other end of the dependence. In this example, coherence is needed only for `A(5)`. The complete picture is what happens to each of the four recognizable sections in each epoch:

Epoch	A(1:3:2)	A(5)	A(7:2*N:2)	A(6:N:2)
1	read	read		
2	read/written	read/written	read/written	
3		read	read	read

We manually apply section-based dependence analysis to our test suite. We assume that can do the following:

- Represent rectangular sections with steps
- Analyze affine subscripts of the form $c_1i_1+c_2i_2+$ ...
- Propagate constants
- Recognize simple, scalar, symbolic equality, i.e. know that 'N' = 'N'

These are all standard, conservative assumptions and not presumptuous about what is available in dependence analysis nor about what will be available in section analysis. The above example is about as complex as we presume to be able to analyze.

9.1.3 Coloring

The next step is to feed this analysis to the coloring algorithm, IAColor (§8.5). It currently handles sequential code and loops but not inter-epoch DAGs. We have not encountered any inter-epoch control-flow DAGs in our test suite. As input to IAColor, the above analysis is written in the following trivial syntax:

program ::= stmt*
stmt ::= assign | loop | if
loop ::= 'loop' stmt-list
if ::= 'if' stmt-list ['else' stmt-list]
stmt-list ::= '{' stmt [,stmt]* '}'
assign ::= [var[,var]*] '='[var[,var]*] ';'
var ::= <simple name>

The important aspect of this trivial language is that every section which is recognized in dependence analysis has a unique name and IAColor is not concerned with what the section looks like. This syntax captures everything of interest to TS'[3].

---

[3] If there were insufficient colors, it would be useful to augment this with graph weighting.

IAColor parses the program and optimally (§8.5.2) colors the program. It returns the location of invalidates and the color for each invalidate and array reference. In our test suite, four colors were sufficient to color every program without conflict on a unit cache line machine. Five colors were needed for the non-unit cache line case on LU (the rest still took only four).

9.1.4 Program Augmentation

The output of IAColor is simply a listing of where to place invalidates and how to map colors to those invalidates and references. This must be manually transformed into program code. The mapping process comes in two steps:

1) Assigning regions (analytically determined sections) to ranges (run time recognizable sections, §8.6) .
2) Adding the directives that manipulate the color of ranges.

For the test suite, step (1) was done by using only the implicit RLB mechanism (§8.6). This was adequate for every case. All of the regions in the test suite were simple ones which mapped directly to a single range. The complexity of the suggested heuristic (§8.7.4) was not needed. The manual process was therefore trivial and not presumptive of any special algorithmic ability.

The actual code changes in step (2) allocate RLB entries, map colors to those entries, and place invalidates as needed. The following directives are used for TS':

1) SetColorRange(*region number*, *start address*, *stop address*, *step*)
2) MapRegionToColor(*region number*, *color*)
3) Invalidate (*color*)
4) MaxColor (*maximum number of colors for all cycles*)
5) CycleRegion (*region number*) – subsequent references to this region will use the next color in the cycle.
6) CycleInvalidate – subsequent invalidates will use the next color in the cycle.

The other local strategies simply use:

1) Invalidate (*start address*, *stop address*, *step*)

For instance, after all of these steps, the initial example (figure 9.1) would become:

```
call SetColorRange(1,A(5),A(5),1)
call MapRegionToColor(1,red)
PDO I=1,5,2
   =A(I)
ENDDO
PDO I=1,N
   A(2*I)++
ENDDO
call Invalidate(red)
BARRIER
PDO I=5,N
   =A(I)
ENDDO
```

The tasking model is that all processors execute the code. `PDO` is a work sharing construct that splits up work. Each processor executes the statements outside of a `PDO`. The `SetColorRange` call allocates an RLB entry for a section, in this case a section consisting of the single element `A(5)`. `MapRegionToColor` sets the initial color of that section. The `Invalidate` then resets all of the ‘red’ bits which can be set only for `A(5)` in this case. Finally, each processor executes the `Invalidate` once then they all rendezvous at the `BARRIER`. The important aspect of this is that IAColor automatically specified the `MapRegionToColor` and `Invalidate` calls. Only the `SetColor-Range` was a result of hand dependence analysis.

9.2 Cache Organization Assumptions

Coherence policies are orthogonal to many of the usual cache organizations properties. Cache associativity does not affect the correctness nor the implementation of local coherence strategies. It might still impact their effectiveness. We present data to indicate that it does not affect the relative benefit of coherence strategies.

Write policy is also unrelated to the coherence strategy. It can be handled in any of several ways. It could be write-through. So long as writes are guaranteed to complete to main memory in some fixed number of cycles, such that they will not still be pending across a barrier, write pipes can be used and stalls largely avoided. Note that this is not sequential consistency [43]. Writes can complete to memory out of order. Since parallel loops are guaranteed to be data-race free, this is not a problem. Nor is this even release consistency because it says nothing about invalidates or completion with respect to other processors, only with respect to main memory.

The other possibility is that write back is used. In this case, the writes need to guarantee to complete to main memory before a barrier finishes. This can be arranged in hardware in some special way. It could be handled trivially by flushing dirty words to memory just before the barrier (though that could create contention problems). If hardware provides no assistance, it could be handled via a CTV-like mechanism for handling updates (chapter 5). Regardless, none of these strategies require global communication.

9.2.1 Cache Size

For most of the tests the cache size was set sufficiently large that the tests ran completely in cache, i.e. there were only cold start and coherence misses, not capacity misses. Thus, the tabulated miss rates overstate what can be expected in a real program. The purpose of this data is not to make any determination of what miss rates should be expected of the programs in the test suite. Rather, the purpose is to explore how much of the available inter-epoch reuse can be captured by coherence strategies. We previously addressed the question of there being adequate inter-epoch reuse to even be worth pursuing (§3.1.1).

Eliminating capacity misses exposes the difference in the effectiveness of competing coherence strategies while minimizing the effects of particular machine configurations. Limited cache size does not change the relative order of different coherence strategies' miss rates. It merely limits the number of misses for which improved coherence strategies can have any effect. We present data on its impact later in this chapter (§9.4.8).

Similarly, only shared data is reflected in the statistics. Private data is excluded since it can always be a cache hit and does not affect the miss rates.

Realistically, we were limited to simulating 128K caches. Because of the simulator overhead structures, larger cache sizes created simulation executables that were too large to execute. Similarly, the test runs were becoming too long when executed on such large problem sizes. The sizes of our test cases were generally chosen as large as possible within these constraints.

9.2.2 Cache Line Size

As already noted, coherence must be preserved for each word individually (§8.8.1). This “logical” cache line is not the same as a physical cache line, nor does it constrain the physical cache line size. Insisting on unit logical cache lines does not produce the same effect as using unit physical cache lines. An eviction with a physical non-unit cache line still evicts all words in the physical line and a miss loads all of the words in the cache line, even though logically the cache state is per word. A physical unit cache line would evict only the word in question.

Since local strategies must operate at this finer granularity, they also gain some benefit from the finer granularity because an invalidate might affect only part of a cache line. To make a fair comparison, the global strategy was also modified to have unit logical lines so that its invalidate logic also operates on the finer granularity.

The improved miss rate that comes with non-unit cache lines is because of spatial locality. That comes with the longer physical cache line, regardless of the logical cache line size. Local strategies requiring unit logical cache lines does not compromise their ability to profit from non-unit physical cache lines.

Our tests were run with both unit physical cache lines and four-word physical cache lines. The results were proportionally similar in both cases. The four-word cases confirm that coherence strategies are not line size sensitive. The unit cases show better resolution on the actual effectiveness of the strategies (since 4 times as many accesses are subject to coherence).

9.2.3 Scheduling

Most tests were run with a block iteration schedule for parallel loops (where each processor gets a set of consecutive iterations). Cyclic distributions are also examined. In no way do the coherence algorithms ever depend on this.

Load balancing strategies try to better utilize processor time based on non-constant execution time for an iteration. In that effort, they usually choose a 'random' iteration, random in the sense that it has no relation to the underlying dependence pattern. Such choices lower the overall miss rate but do not affect the relative relation of the strategies. We have not examined these in detail.

9.3 Programs

We studied the miss rates for four programs [Appendix B].

Program	Lines of Code	Tries	Colors: Cache Line Size: Unit	Colors: Cache Line Size: 4 Word
LU:	200	n/a	4	5
Heat Flow:	200	655	3	4
Erlebacher:	1300	1609	4	5
Erlebacher:	1300	3402	forced 3	n/a
FFT:	150	14	2	2

Table 9.1: Test Suite Summary

Table 9.1 summarizes the characteristics of the test suite. 'Lines' is simply lines of source code (including coherence operations). 'Tries' is the number of test colorings examined by IAColor. 'Colors' is the number of colors needed to avoid all conflicts in the coloring graph. To study how TS' degrades when insufficient colors are available, some cases were tried with deliberately insufficient colors. These are listed as "forced".

9.4 Analysis of Miss Rates

Miss rate data alone cannot answer every question about comparing coherence strategies because it does not take into account the run-time execution overhead of different coherence operations nor the cost of building some of the required hardware. However, miss-rate data can address several important questions:

1. Can local strategies ever be adequate? *Yes*
2. Does the improved resolution of section analysis matter? *Yes*
3. Are dynamic strategies significantly better than static? *Yes*
4. How many hits must TS' sacrifice? *Depends on the program*

The effectiveness of different coherence strategies is determined primarily by the dependence pattern of the program to which they are applied. As we expect and the data supports, parameters such as problem size, number of processors, cache size, cache line size, and cache associativity have little impact on the relative effectiveness of coherence strategies. In the presentation of this data, we instead concentrate on the difference in performance based on coherence strategy and program type.

FSI data reflects write-misses being as expensive as read misses (§4.2.1). We address this in more depth later (§9.4.7).

We intentionally focus on the miss rate rather the hit rate because the misses represent the real cost that is being minimized. The visual difference between a 90% hit rate and a 95% hit rate looks small. But, the actual performance difference will approach a factor of two for slow main memory (e.g. a memory bound program on a machine with memory 50 cycles away [19] would show a speed up of 71%). Thus, a 10% miss rate next to a 5% miss rate better represents the actual improvement.

9.4.1 Ideal Local Versus Optimal Global

On analyzable programs, ideal local strategies perform identically to global strategies (figure 9.2). On Erlebacher, Heat Flow, & FFT, TS1 achieved exactly the same miss rate as the oracle global strategy. In each case, the compiler was able to predict exactly what would be written (though not where). That information plus the local schedule theorem (thm 3.5) was sufficient capture all possible reuse (§3.2.4).

For LU, TS1 did not do quite as well as a global strategy. The reason is the conditional around the swap (§B.2). The relevant parts of the TS1 version are:

```
if (imax <> k) then
   swap a(imax,1:n),a(k,1:n)
end if
…
INV a(imax,1:n)
INV a(k,1:n)
FORK
```

The problem comes when `imax=k`. In that case, no swap occurs, but the invalidate occurs as if it does and unnecessarily removes that data from the cache. This is an effect of the presumed strength of the dependence analysis rather than of TS1 *per se*. In this case, it is reasonable for dependence analysis to recognize that `imax` is unmodified between the swap and the end of the epoch. It can simply be propagated as a trivial symbolic to the invalidate. If dependence analysis were strong enough to understand the guard around the swap and propagate that to the invalidate, TS1 would produce:

```
if (imax <> k) then
   swap a(imax,1:n),a(k,1:n)
end if
…
if (imax <> k) then
   INV a(imax,1:n)
   INV a(k,1:n)
end if
FORK
```

Such aggressive analysis is not common today and we report data for the more pessimistic case. But if were available, the LU code would again be optimal for TS1 and achieve exactly the same miss rate as a global strategy.

9.4.2 TS1 Versus TS

Array sections do matter. There is significant reuse to be found by worrying about which parts of arrays are actually accessed as reflected by the difference between the TS1 and TS miss rates. TS is an ideal local strategy applied to an analysis that is whole array (any write to an array is treated as if the whole array were written). TS1 is ideal at any resolution, including the best available.

Erlebacher (§B.3) was the most extreme example of this in our test suite. Preserving inter-epoch reuse improved its miss rate by a factor of just under 3. The following three loops are an example:

```
20    f(2:N-1,*,*)=
…
50    f(N,*,*)=
…
70    f(1:N-2,*,*)=
```

For TS whole-array granularity of analysis, loop 50 appears to write to all of `f`. The result is that no reuse from loop 20 can reach loop 70, even though nearly all of the array is available for such reuse. TS1 gets full reuse between loop 20 and loop 70.

Heat Flow (§B.1), in contrast, modifies almost all of the array any time that it modifies part of it. However, the border is only read. Once it is in cache, it can stay there. TS1 correctly handles this. TS forces the border to be reloaded. The number of accesses for which that applies is only $O(n^{1/2})$ of the total accesses, but for a 50×50 array that still produced a 20% penalty for TS versus TS1.

For LU Decomposition (§B.2), the situation is more complicated, but essentially an effect similar to that in Erlebacher is reflected in the data. The row swap accesses only a fraction of the data, but it causes TS to lose all reuse around it. This is the reuse from reducing the main block on one outer iteration to the next. Losing all of that caused TS to perform almost as badly as FSI.

The FFT computation (§B.4) has poor processor locality. The complex subscript function causes data to more likely be accessed on a different processor every time that it is accessed. This is reflected in the high absolute miss rate as well as the relatively close miss rates of TS, TS1, & a global strategy. As in the LU case, TS1 was able to use a simple symbolic in the invalidate so as to catch precisely what was written. This lets it perform equally to a global strategy while capturing some 'accidental' reuse that TS sacrificed.

9.4.3 Static Local Versus Dynamic Local

For analyzable programs, those where TS1 does as well as a global strategy, the difference between FSI, a simple static strategy, and TS1, an ideal dynamic strategy, is the amount of inter-epoch reuse in the program. This gap reflects on whether coherence is even an interesting question (for epoch-based parallelism). If the gap is small, i.e. if programs have little available inter-epoch reuse, there is no need for coherence. One can simply use a cheap strategy like FSI to solve the problem. Our data shows that with adequate cache, inter-epoch reuse is worth pursuing.

For Erlebacher, there was a four-fold improvement between TS1 & FSI. This was mostly for the same reason that TS1 did better than TS on Erlebacher: loop 70 referenced nearly the same data as loop 20 and with a similar schedule.

Heat Flow also had nearly a four fold improvement. In this case, it is easier to see why. Consider a 50×50 array with 5 processors sharing the work using a block distribution. Each processor gets 10 columns. The stencil reaches one column left and one column to the right of the element being updated. So long as one column will fit in cache, it does not matter what the actual stencil looks like. In FSI, a given processor needs to read 10 columns plus 2 on each edge and write one column. All of the rest of the reads from the stencil will be hits (give or take some boundary conditions). However, for TS1, only the borders represent real communication. The write of `Grid2` can be a hit from the read in the previous epoch and the read of `Grid1` can be a hit from the write in the previous epoch. Therefore, TS1 needs only to miss for 2 columns worth of data instead of 13. The actual details keep the final results from being so simple, but this observation is the main reason that TS1 is 4 times better than FSI.

LU has a complex sharing pattern. Since the sub-block being reduced shrinks for each iteration of the main blocking loop (`kk`), the distribution of data to processors keeps changing. The natural miss rate is fairly high because of this. Still, there is much incidental reuse. By exploiting this, TS1 does nearly twice as well as FSI.

FFT has a complex and chaotic sharing pattern (from a compiler's perspective). The data really is moving to different processors every epoch. The computation affects the whole array before any part is reused. TS1 improved on FSI only to that extent that chance would leave an element on the same processor the next time it was referenced. There is little for coherence to achieve on FFT-like programs.

We also looked at LSS. It consistently performed worse than TS and better than FSI. Since it too is a dynamic strategy like TS, it makes for an uninteresting data point (it's sole advantage is a cheaper hardware implementation than TS).

9.4.4 TS'

TS' has the same miss rate as an oracle global strategy when there are sufficient colors and all relevant scalars are known when coloring the essential access. For instance, it did *optimally* (as well as a global strategy and not merely *ideally*) on Erlebacher and Heat Flow with 4 colors. For instance, when applying TS' to Heat Flow (§B.1), the first three colors (arbitrarily designated red, green, & blue) were used for the cyclic coloring for the interior regions that are being rewritten. The final fourth color, yellow, was used for the border without any change. The border region is determined solely by the size the problem to be solved. This parameter is known before the main timing (`t`) loop begins. Thus, the analysis has sufficient information to assign a color to the region at the time of the essential read. This all yields essentially:

```
MapRegionToColor(Grid1.border,yellow)
MapRegionToColor(Grid2.border,yellow)
SetColorRange(1,Grid1.interior)
MapRegionToColor(1,red)
SetColorRange(2,Grid2.interior)
MapRegionToColor(2,green)
MaxColor(3) // (red,green,blue) form a cycle, not yellow
DO t
   Grid1.interior = f(Grid2.interior,Grid2.border)
   Invalidate (green) // will be green, blue, red...
   CycleRegion(1) // cycles color for Grid1.interior

   Grid2.interior = f(Grid1.interior,Grid1.border)
   Invalidate (red)
   CycleRegion(2)

   CycleInvalidate // implicitly change color of subsequent invalidates
```

The situation with Erlebacher was similar. However, in LU, the following is problematic for TS':

```
...=a()...I // previous iteration reductions
imax =...
if (imax <> k) then
   swap a(imax,1:n),a(k,1:n)
end if
...a()... // subsequent reductions
```

There is an AT that has its essential write at the swap. The essential read of the AT occurs before `imax` is known. In TS', the color must be known at the essential read. Since `imax` is not known at that point (previous iteration reductions), it must be assumed to be anything. The swap then appears to write all of `a`. This approximation causes TS' to have some unnecessary misses. TS' did properly handle the sections in later reductions. For this test case, it captured 40% of the reuse that an ideal local strategy could have versus TS, leaving TS' net 14% better than TS.

TS' had the same problem on FFT that it did on LU, the section could not be known at the time of the essential read. In this case, it performed only as well as TS. Four colors were still sufficient that it did no worse than that. In any case, there was very little benefit to be gained on FFT, only 2% between TS and global. That TS' failed to realize this benefit is not significant. This is not a coincidence for this code, but what we expect in general: when TS' does no better than TS, there is very little gain to be had.

TS and TS’ share this latter problem, the decision about what to do must be made at the essential access. Therefore, for TS’ to do worse than TS would require a case where four colors are insufficient to color a whole-array only coherence graph. That is, TS’ could in principle be applied to a program that uses the same analysis as TS, whole-array only. So long as four colors are sufficient there, insufficient colors on a section-wise coherence graph will do as well or better. It would be unusually complex program that requires more than four colors at the whole array level. We have not seen any such program. TS’ will have a higher hit rate than TS, in principle not just as an empirical result, on all but the most unusual of cases.

9.4.5 TS' and Irregular Problems

The application of TS’ to irregular problems is a limiting case worth exploring separately. By irregular we mean a program which shuffles its data between processors every epoch in such a way that no simple schedule (e.g. cyclic or block, and not a custom schedule just for the problem) will capture more than a random amount of reuse. We also refer to the parts of programs which are irregular even though the entire program may not be. Consider this particular limiting case of a loop with irregular data access:

```
DO I=1,M
   PDO J=1,N
      if (.not. q)
         A(f(J)) =…
         …= A(f(J))
```

where q is an expression which is true q% of the time and f is a random permutation of `1..N` that changes for each iteration of the `I` loop. The read of `A` represents 50% of the references and will hit every time. This is the intra-epoch contribution to the hit rate that every coherence strategy gets, call it r. Ignore the intra-epoch hits for the moment. For a global strategy the probability of a hit is the chance the same element was referenced on the same processor the last iteration of the `I` loop, which is $1/p$ where p is the number of processors, plus the chance no processor referenced it last epoch and it was referenced on the same processor two epochs ago, etc. If the element were referenced last epoch on a different processor, then it would be stale and there would be no reuse. Global strategies can look back arbitrarily far (assuming infinite cache).

Since f changes for every iteration of the `I` loop there is no chance that TS’ will be able to analyze the inter-epoch sharing. It will behave like TS. Even with no useful analysis, both strategies will preserve reuse between two adjacent epochs. TS' looks back exactly one epoch. CTV+ can look back one epoch, half the time. Thus the probabilities of an inter-epoch hit are:

- $FSI = 0$
- $$\begin{cases} Global(p,q,M) = \frac{1}{M} \overbrace{\sum_{I=1}^{M} \overbrace{\sum_{i=1}^{I-1} q^{i-1} \frac{1-q}{p}}^{\text{a hit } i \text{ iterations ago}}}^{\text{summed over all iterations}} = \frac{1}{p} - \frac{1-q^M}{pM(1-q)} \\ \lim_{M\to\infty} Global(p,q,M) = \frac{1}{p} \end{cases}$$
- $$\begin{cases} TS'(p,q,M) = \frac{1}{M} \sum_{I=1}^{M} \sum_{i=1}^{\max(1,I-1)} q^{i-1} \frac{1-q}{p} = \frac{M-1}{M} - \frac{1-q}{p} \\ \lim_{M\to\infty} TS'(p,q,M) = \frac{1-q}{p} \end{cases}$$
- $CTV+(p,q,M) = \frac{TS'(p,q,M)}{2} = \frac{M-1}{M} * \frac{1-q}{2p}$

FFT (§B.4) approximately fits this model. Each pass of the `K` loop halves the length of the section which each processor operates on in the parallel loop making the inter-epoch reuse 'random'. The q factor arises because for non-power-of-2 sized arrays some elements will be skipped when halving `incrm` rounds down. We applied this model to FFT with unit cache lines and a cyclic processor distribution on one power-of-2 and one non-power-of-2 case (table 9.2). Its predictive power was fairly good, missing by only 4 hits for the global and TS' case. It missed by more on the CTV+ case because there were only 5 epochs (M) of which CTV+ preserved inter-epoch reuse on 3. The above formula was modified to have a factor of 0.6 instead of 0.5 but that didn't quite compensate because of the way the changing number of applied processors interacted with exactly which epoch boundary was preserved. These reasons do not apply to the TS' & global cases and don't need to be compensated for there. A block-distributed FFT does not fit the model well because the inter-epoch reuse rate is significantly higher than $1/p$, but this factor applies to local and global strategies equally. The following observations hold for block distributions as well.

Unit Cache Line FFT

		Size: 50	Size: 32
M		5	5
p (average)		3.4	3.4
q		0.096	0
references		565	400
intra-epoch hits, *r*		339	240
initial references in parallel loop		226	160
Expected Hits for initial references	Global	52	38
	TS'	48	38
	CTV+	29	23
Expected Hits (total)	Global	454	318
	TS'	445	318
	CTV+	419	295
Actual Hits	Global	450	314
	TS'	445	314
	CTV+	431	302
Difference	Global	4	4
	TS'	0	4
	CTV+	-12	-7

Table 9.2: TS' Applied to FFT

There are three important observations, all of which we expect to be true for irregular programs in general. First, if q=0, a global strategy has no benefit at all over even a simple local strategy. This can be generalized beyond the irregular case: that part of the shared data which is written in every epoch cannot gain any benefit from a global strategy. Second, for small q, local strategies do nearly as well as global strategies. This reflects the fact there simply is not much inter-epoch reuse to be had.

Third, even if (as in the above example) an element is only referenced twice per iteration on average, the intra-epoch hit rate will account for at least p times as many hits as the inter-epoch reuse, leaving local and global strategies with nearly the same performance. As the number of processors increases (presumably as part of scaling up the problem size), the gap between a local strategy and a global strategy vanishes, because the intra-epoch reuse stays the same, or nearly so. This is in contrast to a more regular program where the inter-epoch reuse will not fall so rapidly with p.

If q is large, that means the computation is sparse and the amount of data cached continues to grow with the number of epochs. Such a problem may still fit mostly in cache and global strategies will derive some benefit as just discussed. Recent references are more likely to be reused than old references at the local, intra-epoch, level. If this property holds for a program at the inter-epoch level, the original assumption about f being a uniform distribution is a poor one. It will be weighted toward more recent epochs and lessen the extra benefit that global strategies gain by looking backward.

9.4.6 K&S

We directly compare K&S (§2.2.2) only to TS’ because both are aggressive, particular implementations of abstract strategies. They are each the possibly best competitors from their respective classes of algorithms. TS’ is a particular implementation of an (almost) ideal local strategy. K&S is a VM-based implementation of a global strategy. Since it is software instead of hardware, it is possible to get an accurate measure of its overhead costs be measuring the extra number of memory accesses it requires to implement its protocol. We chose it for this reason. Getting an accurate measure of a true hardware global strategy is considerably more difficult. The latter will usually have lower run-time overhead costs than K&S, but not arbitrarily so. At some level, a certain amount of extra traffic is essential to maintaining global coherence. Measuring the K&S strategy gives us a handle on that.

For each test case, we measured K&S across a range of possible page sizes. For the most representative Erlebacher scenario, K&S performed best with a 128-byte page size (fig. 9.8). With a larger page size, false sharing caused over-invalidation. Invalidates reach into other regions because of the minimum granularity. With smaller page sizes, the miss rate of K&S improves, converging with that of an oracle global strategy since it is a global strategy, but the net cost goes up because each stale line requires its own invalidate message, thus losing the benefit of aggregating coherence information per page. For K&S each invalidate message is itself a global memory access. We count those the same cost as a data-cache miss induced by a global access.

The optimal page size was either 128 or 256 bytes in our tests, for all problem sizes and all programs. In all cases, it showed the same basic shape as in the Erlebacher example, a curve with a single minima in the middle. Instead of trying to determine the best operating point for K&S, we simply use its best performance on every test case independently as the comparison point in our data.

Looking at the actual data (principally figure 9.2), the overhead cost (versus oracle global) runs from 21% for Erlebacher to 83% for FFT. In those cases where TS' can perform optimally, e.g. Erlebacher, this is a serious penalty. In Erlebacher and Heat Flow the data has a high probability of being accessed again on the same processor in subsequent epochs. In these simple and regular cases, the K&S overhead was low. This is the expected result since the cost is proportional to the number of invalidates. For FFT and LU, the sharing is more irregular. The higher number of real invalidates caused K&S overhead to be higher. As a percent of the miss rate, the K&S overhead is about the same for Heat Flow & LU, but the work K&S does is over all accesses. Versus the hit rate, it would be clearer that the K&S overhead is higher for LU & FFT.

K&S does well when there is little actual sharing, regardless of the analysis. TS' does well when the analysis can show that access is regular. If the program is irregular at runtime because of scheduling, TS' will do as well as an oracle global strategy (since there is no benefit to be gained) and K&S will do at least some worse because of the invalidate overhead. This holds regardless of the quality of the compiler analysis. If the program is analytically regular and runs with a high degree of inter-epoch reuse, K&S overheads will be low, but TS' will capture all of the reuse with no extra global overhead. It is only when the actual computation has a lot of inter-epoch reuse but the program is not analytically regular that K&S does comparably to TS'. The closest example is LU (figure 9.9). TS' sacrifices about half of the available reuse (between TS & TS1) because of the imprecision of the analysis. K&S caught most of that extra margin (between TS' & TS) for the raw misses. But, its overhead was high because the actual sharing pattern was somewhat irregular (with shifting block distributions on each iteration of the outer blocking loop). On net, TS' still did better.

To understand where K&S does well, consider the following artificial code fragment designed to exaggerate K&S's strengths. Assume it runs with two processors and processor 1 always getting iteration `j=1`:

```
do i=1,N
   do k=1,M
      PDO j=1,2   -- loop 1
         if (pid = i mod 2) then count++
      end do
      PDO j=1,2   -- loop 2
         if (false) count = 0
      end do
   end do
end do
```

The data gets reused on every iteration of the k loop and then migrates processors on every iteration of the i loop. The program is such that local strategies cannot realize any inter-epoch reuse. They preserve reuse from loop 1 to loop 2, but there is none. The conditional confounds analysis making it seem that ia was written in loop 2. Local strategies cannot find the reuse on the k loop. They miss all of the N*(M-1) possible inter-epoch hits. An oracle global strategy would catch every one of those (leaving only the 1 cold start and N-1 obligatory migrations) . K&S can preserve the k-loop reuse but it takes two iterations of the k loop for the *weaklist* to be cleared out (explained in appendix C). Therefore, K&S gets N*(M-2) of those hits. It's overhead is (in this case) equal to the number of misses. It effectively gets ≈N*(M-4) of those hits. If the data frequently migrates, i.e. M<5, K&S offers no advantage over a local strategy.

<u>Scaling</u>

For programs where the number of copies of a cache lines increases as the problem scales, K&S will have superlinear coherence costs. Similar costs will apply to any global strategy which attempts to track all copies of a cache line. We examined the scaling behavior of Heat Flow (figures 9.10 & 9.11). The first figure scales by keeping the data per processor constant and the second scales by keeping the number of rows per processor constant (while the total data per processor grows linearly). We regard the former as more natural here since the miss rate for a global strategy stayed reasonably constant. The per reference cost of K&S increases linearly with the problem size in this case (this includes misses and obligatory coherence references). In the second case, the per reference cost of K&S also continues to increase, but in a less obvious fashion.

For LU & FFT, as well as Heat Flow, the miss rates for TS1 (or global) and K&S fit a linear model under the constant data per processor scaling scenario. FFT, similar to Heat Flow, showed increasing costs for K&S. For LU, the K&S miss rate minus the TS miss rate decreased slowly. The K&S/TS1 ratio falls in this case, but simply because the K&S

miss rate stays a nearly constant amount worse. Erlebacher's cubic behavior makes the range of sizes we can simulate too small to find a clear scaling trend. The limited data shows TS1 and K&S to both have a constant miss rate.

Extrapolating any of these trends is difficult. Clearly, the TS1 miss rate has 100% as a maximum. Therefore, it cannot be linear except over short ranges. The K&S effective miss rate can be > 100% because the extra cost of coherence can get arbitrarily worse as the number of processors increases. When the TS1 miss rate stays constant and the K&S rate increases linearly, the linear rate of increase may cease but we do not expect the miss rate to ever improve because the number of copies of a given cache line will not asymptotically drop as the number of processors increases.

9.4.7 CTV , CTV+, & FSI

For static strategies, the question is which approach yields the best effective way to estimate the misses on a typical mix of programs. Unlike in the dynamic case, there is no firm standard to compare against.

The first observation is that CTV had little gain over FSI. In practice, it can only use its extra resolution to benefit for accesses on the root processor. Those do have some added value because they are on the critical path, but the difference was not significant in any case. This test suite, if anything, was too favorable to CTV because there were cases that caused it to over-invalidate that did not affect FSI. The value of CTV is only that it requires no special coherence hardware, but it does have significant overheads and these are not reflected in the data.

FSI on Heat Flow requires some special analysis to understand why CTV might do better. For Heat Flow, one of the two grids is only written in a given epoch (and the other is only read). CTV's simple strategy of preceding reads with INVs leaves the write-only array completely alone and performs no coherence on it. This is still true (and correct) for non-unit cache lines. Thus, write misses incur no miss penalty for CTV in this case. FSI is intended to be implemented so that writes are buffered in the network and are essentially zero cost. It then uses write-through. The original design specified only unit cache lines. For the purposes of testing on non-unit cache lines, the assumption in this work is that a write miss in FSI causes loads and is counted as a miss (since it has no analysis machinery to prove otherwise). For the non-unit cache line case (figure 9.2), we regard this as reasonably fair. However, for the unit cache line case, it is more appropriate to regard write misses as irrelevant since there are no extra loads.

The data was collected based on counting all misses, write and read. For non-unit cache lines this is justified because write misses still cause reads to load the rest of the cache line. FSI was originally specified only for write-through cache with unit line size. In that scenario, write misses are essentially free. We considered this as a separate case by examining the read-miss only data on unit cache lines (figure 9.4). The assumption of zero cost writes was applied uniformly to all coherence strategies. In this case, the advantage that CTV has over FSI on Heat Flow vanishes. All other cases stay essentially similar.

When the write accesses within an epoch can be precisely analyzed and invalidated at the boundary between epochs, CTV+ will split the difference between FSI (which gets none of the inter-epoch reuse) and a global strategy (which gets all of the inter-epoch reuse) because it lets every other epoch 'pass' without invalidation. For instance, the miss rates for the read-miss only case of Heat Flow (figure 9.4) are:

FSI – 24.9%
CTV+ – 14.1%
Global – 3.6%
FSI - CTV+ = 10.8%
CTV+ - Global = 10.5%

Heat Flow is completely analyzable because of its simple sharing pattern and the expected result is observed.

For LU, the analysis is similar to the TS1 case. The imprecision of the analysis causes CTV+ to over invalidate. It caught 26% of the gap between FSI and Global instead of the maximum of 50%.

In the case of Erlebacher, all of the sections were simple and analysis was capable of separating all of the border conditions so that invalidates were exact. CTV+ recovered 64% of that gap, better than expected. The reason is that it can analytically collapse subsequent read epochs into one (§6.1). Consider the last row of the main data array, `uud`, in isolation. Its access pattern looks like:

```
10 uud(N,1:N,1:N)   =…
50 uud(N,1:N,1:N) -=…
   INV (uud(N,1:N,1:N))
60 …=uud(N,1:N,1:N)
70 …=uud(N,1:N,1:N)
80 …=uud(N,1:N,1:N)
```

This only needs one invalidate, between loops 50 and 60. Since loops 60, 70, & 80 only read `uud`, there is no need for more invalidates. Therefore, with four opportunities for inter-epoch reuse, CTV+ captures three of them. FSI loses reuse at every epoch boundary because something else is written in each loop and thus change bits needs to be cleared.

For Erlebacher, CTV+ even does better than TS, a dynamic strategy. This holds regardless of the cache line size or whether or not write misses are included. The reason is that loop 20 (not shown) modifies parts of `uud` other than the last row. The limited precision of TS makes the last row valid (not fresh) in loop 20, but it becomes stale before it is accessed again in loop 50 since it appears to be written in loop 20 as well. The above example is just one of the scenarios in which that happens. It could, in principle, also happen between any of the last three loops and suffer the same problem as FSI, though in this particular case there were no writes to `uud` between loop 60 and 70. There were writes to other parts of `uud` in loop 60 and 70, but that alone does not defeat TS. The last row remains valid in loop 70 where it gets promoted to fresh again. If there were reuse from loop 60 to loop 80, but not on the same processor during loop 70, CTV+ would capture that reuse and TS would lose it.

9.4.8 Different Execution Parameters

Cache Line Size

The effect of changing the cache line size from a four-word cache line to a unit cache line is contrasted in figures 9.2 and 9.3 for all test cases (with the latter being scaled exactly 4X the former for a more direct comparison) and additionally for the two-word cache line case on Erlebacher in figure 9.7. The relative effect on each of the different strategies is almost none. The profiles are essentially the same in every case. The difference in height is the average amount of spatial locality captured for each program. For Heat Flow, it was almost complete, the miss rate fell by a factor of 4. It was less for the others. Regardless, each coherence strategy was affected equally.

This is the expected result because spatial locality captures 'near' references which are likely to be intra-epoch reuse, not inter-epoch reuse. Thus, the set of misses which coherence operates on is largely unchanged by different cache line sizes.

There is one case worth noting: TS1 had one more miss than global did on FFT. This was the only observed instance of a pseudo race condition being used to advantage (§8.8.2).

Scheduling

The execution schedule of iterations is an important parameter that can be tuned to trade-off load balance versus cache reuse. An important feature of cache coherence (versus distributed memory) strategies is that they are correct regardless of the schedule. That is, the coherence algorithm does not drive the scheduling algorithm. It is a free parameter which can still be used for other purposes. That still leaves the question of correct at what cost? To address this, we ran the same tests as in figure 9.3 again with cyclic instead of block distributions (figure 9.5). We ran them with unit cache line size so as to directly examine the effects of scheduling instead of also involving questions of strip mining and loop interchange. K&S data was excluded since that strategy is not appropriate to this scenario.

The results were similar to the changing cache line size case, coherence strategies all maintained their relative performance levels. Erlebacher, LU, & FFT all stayed about the same. Heat Flow had a considerably higher miss rate (compared to the other test programs), for all coherence strategies, because most of its sharing comes from accessing the neighbor element in each dimension. A cyclic distribution causes neighboring elements to be on a different processor whereas a block distribution naturally leaves neighbors on the same processor. Still, all coherence strategies were equally affected.

Limited Cache Size

While limiting cache size will not change the relative order of coherence strategies, it can change their relative performance. To get some indication of what can be expected when inter-epoch reuse starts to become unavailable, some of the tests were run with smaller cache sizes and different associativities (which only become relevant when the cache is too small).

For example, consider the following (partial) data from Erlebacher (figure 9.6):

Some Erlebacher Miss Rates with Finite Cache

Cache Size	FSI	TS	TS1	TS 'efficiency'
∞:	12.2%	9.0%	3.4%	36%
8K:	12.6%	12.0%	10.9%	33%

FSI preserves exactly intra-epoch reuse (for Erlebacher). In this case, TS1 did as well as an optimal global strategy. Therefore, with infinite cache there is 12.2-3.4 ≈ 8.8% of the total accesses (but ≈ 72% of the misses) which can be converted to hits by recognizing inter-epoch reuse. TS captured 36% of those. With 8K of cache only 1.7% of the total accesses were possible inter-epoch hits. TS captured 33% of those. The relative efficiency of TS changed only slightly by cache size. Smaller caches simply leave less data available for it to operate profitably on. This is true in general in our test suite, across strategies and programs.

In principle, it could be otherwise. For instance, if the added benefit of TS1 versus TS were to retain data across a greater epoch distance, then decreasing the cache size would lower the performance of TS1 until it equaled that of TS, then both would decline together. The extra data that TS1 retained would be lost first since evictions would tend to remove older data first. Instead, the added benefit of TS1 is to retain sections across near epochs that TS would unnecessarily invalidate.

The same results were observed for all test cases. Decreasing cache size reduced the number of available inter-epoch hits, but impacted all strategies proportionally on that remaining amount.

Program Size and Number of Processors

We also tried varying the number of processors and the size of the problem to see if any trends emerged. As expected, these made no difference so long as increasing the number of processors does not interfere with cache-line based spatial locality. The other parameters affected all coherence strategies proportionally, these parameters just did not make any difference at all. We summarize the results of these parameters plus cache line size and scheduling in figure 9.7 for Erlebacher. The same results were seen for all test programs.

Miss Rates

Blocked, four word cache lines

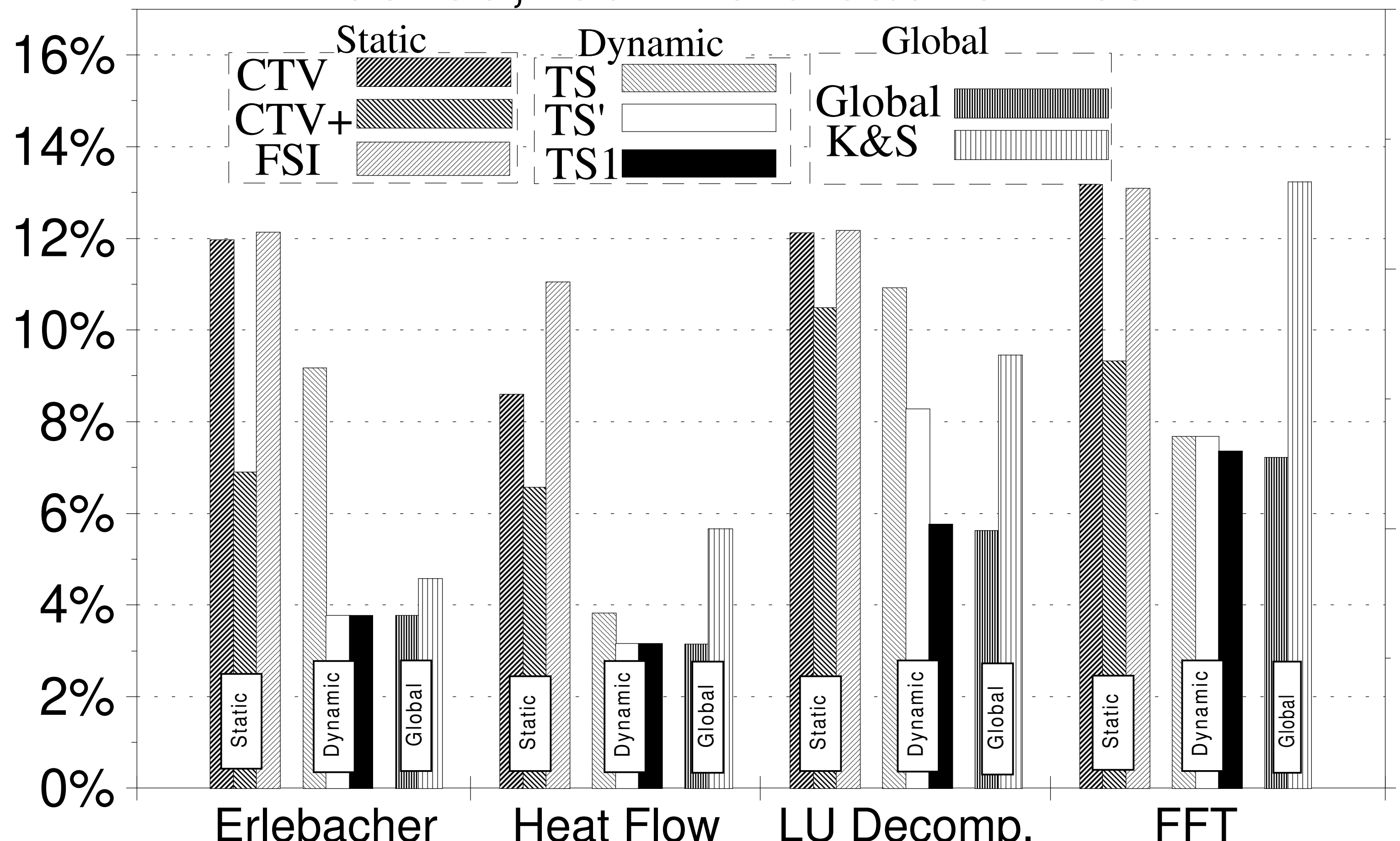


Figure 9.2

For representative problem sizes

Miss Rates

Blocked, Unit Cache Lines

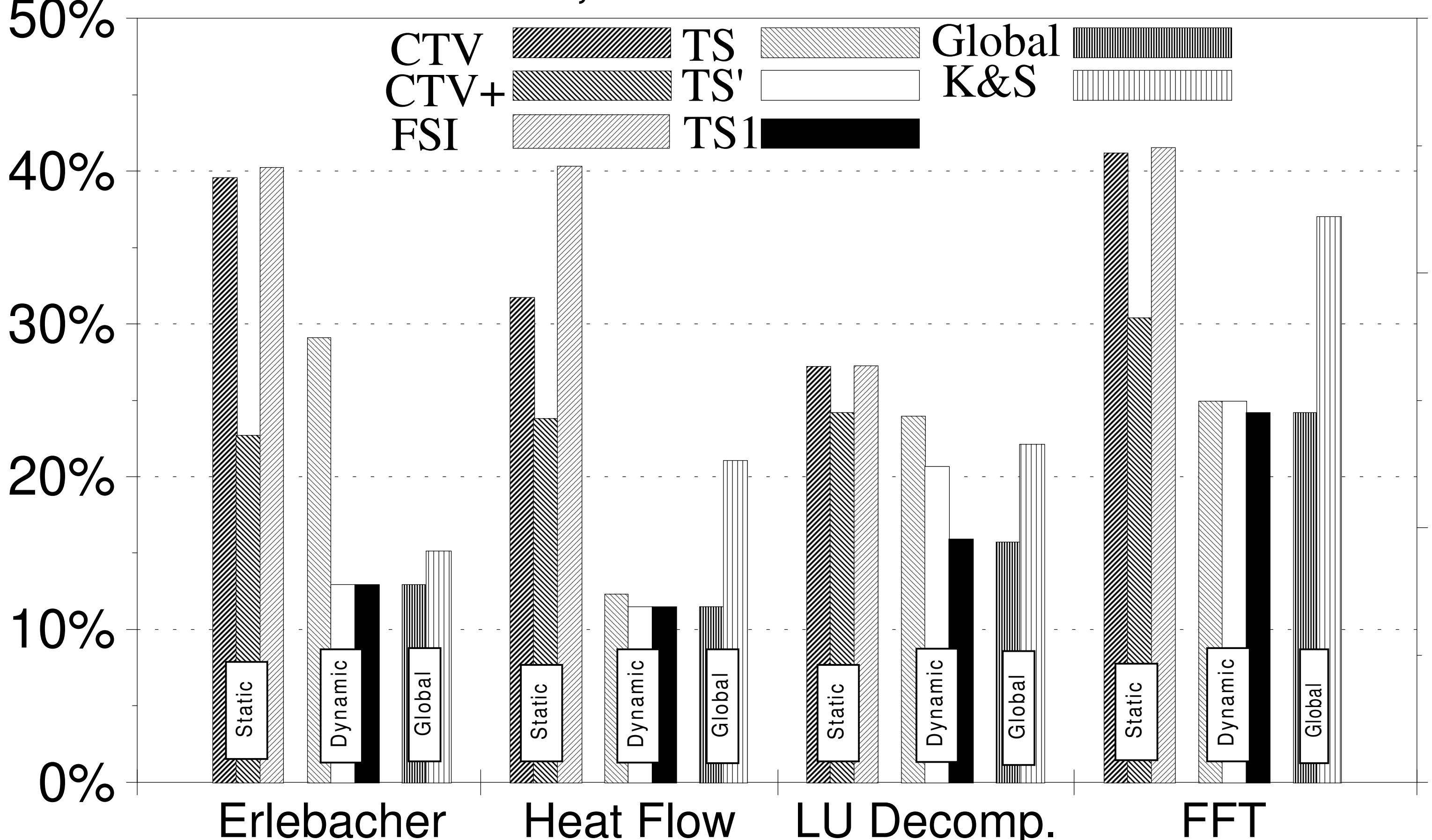


Figure 9.3

For representative problem sizes

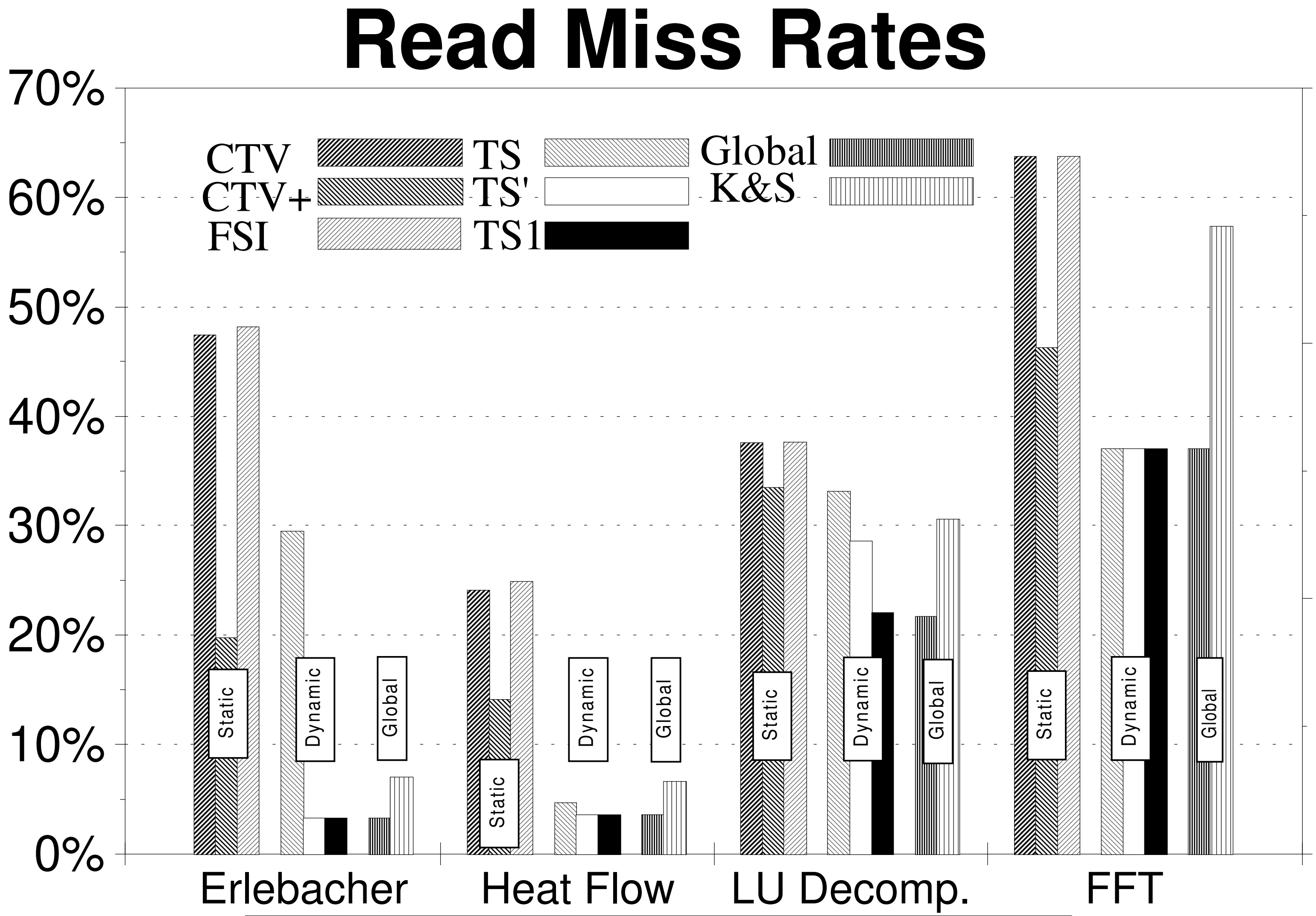


Figure 9.4

For representative problem sizes, Cache Line Size = 1 word

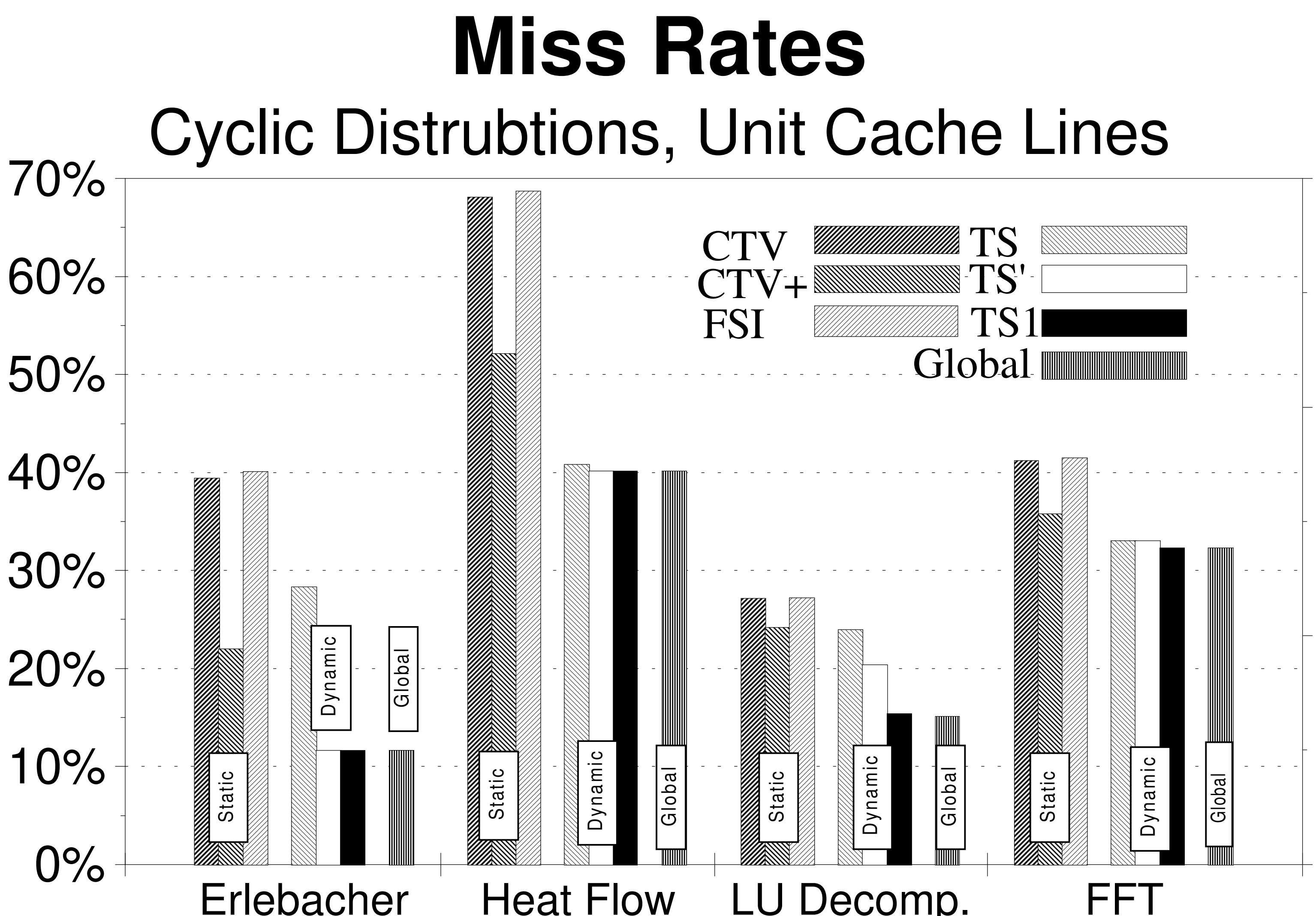


Figure 9.5

For representative problem sizes, Unit Cache Lines, Cyclic Distributions

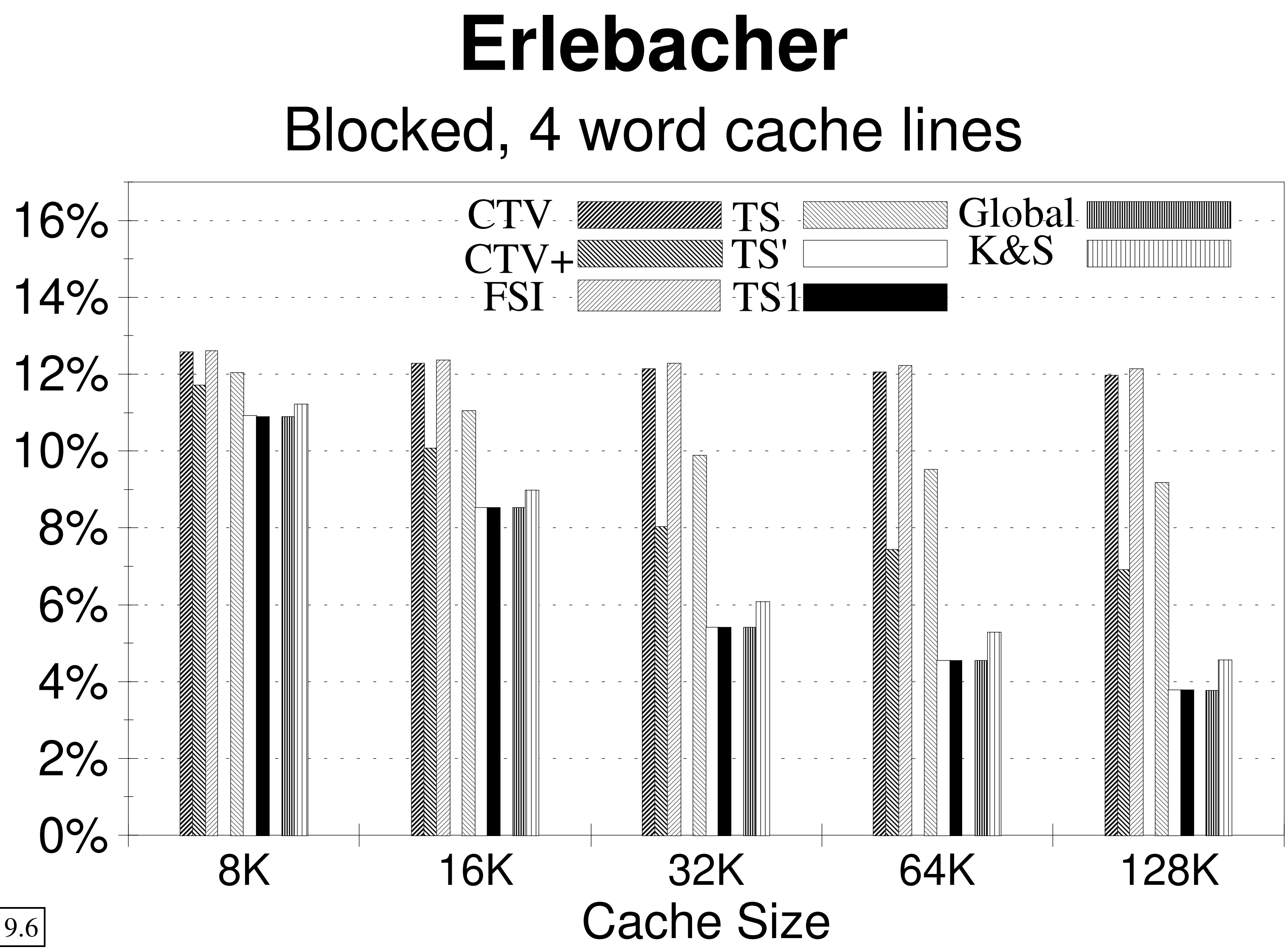

Erlebacher
Blocked, 4 word cache lines
CTV
CTV+
FSI
TS
TS'
TS1
Global
K&S
16%
14%
12%
10%
8%
6%
4%
2%
0%
8K
16K
32K
64K
128K
Cache Size

Figure 9.6

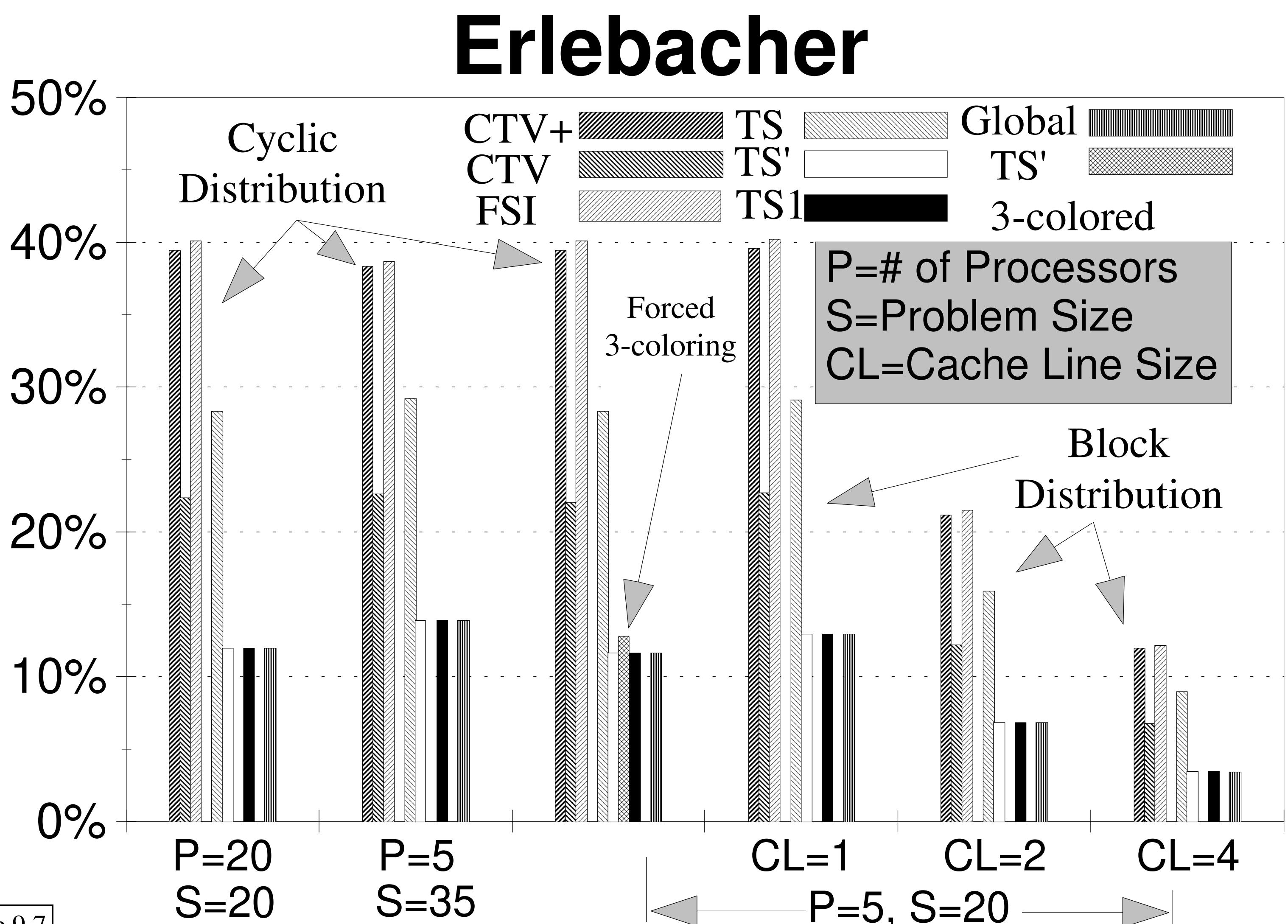

Erlebacher
50%
40%
30%
20%
10%
0%
Cyclic Distribution
CTV+
CTV
FSI
TS
TS'
TS1
Global TS' 3-colored
P=# of Processors
S=Problem Size
CL=Cache Line Size
Forced 3-coloring
Block Distribution
P=20 S=20
P=5 S=35
CL=1
CL=2
CL=4
P=5, S=20

Figure 9.7

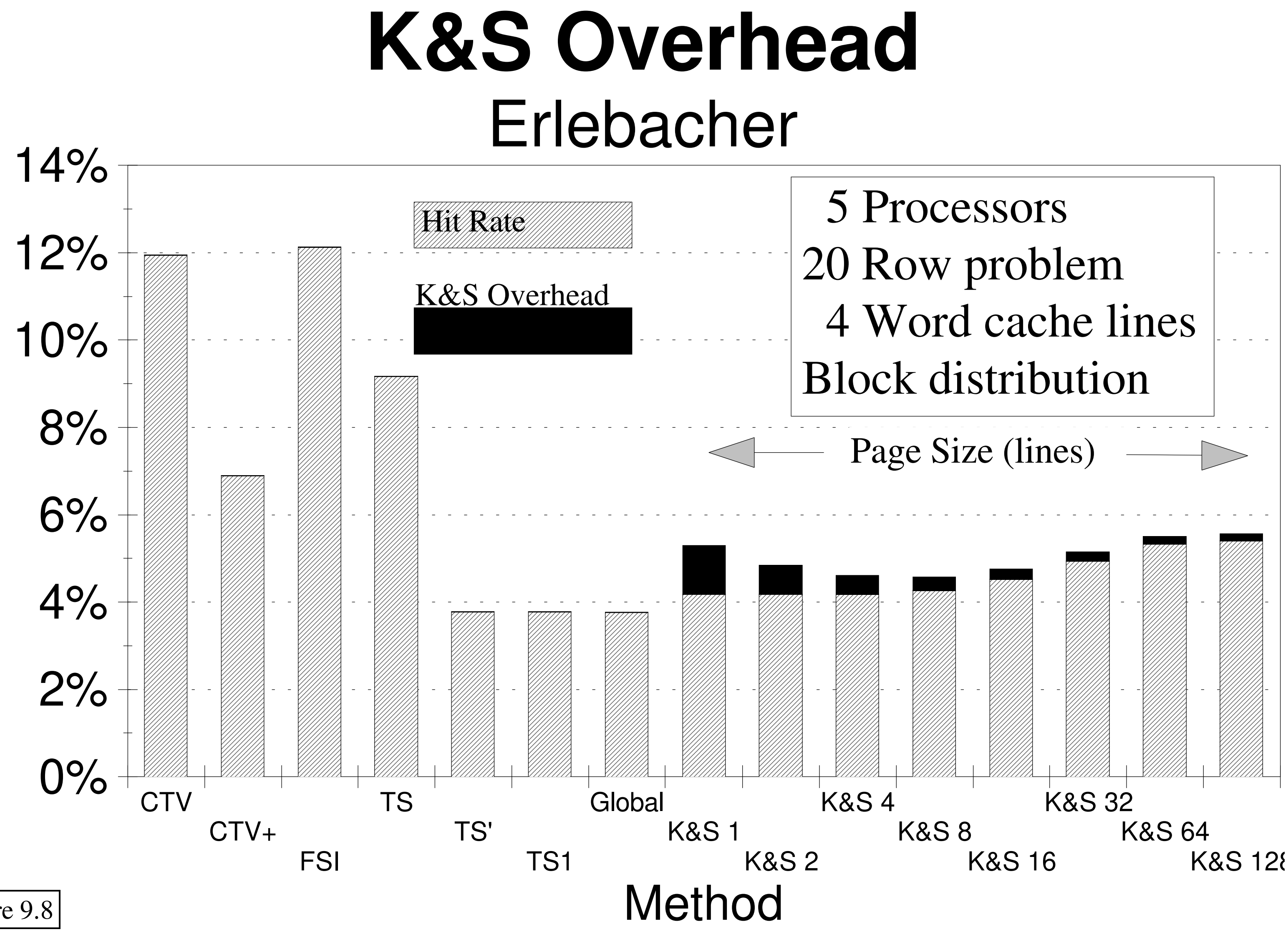
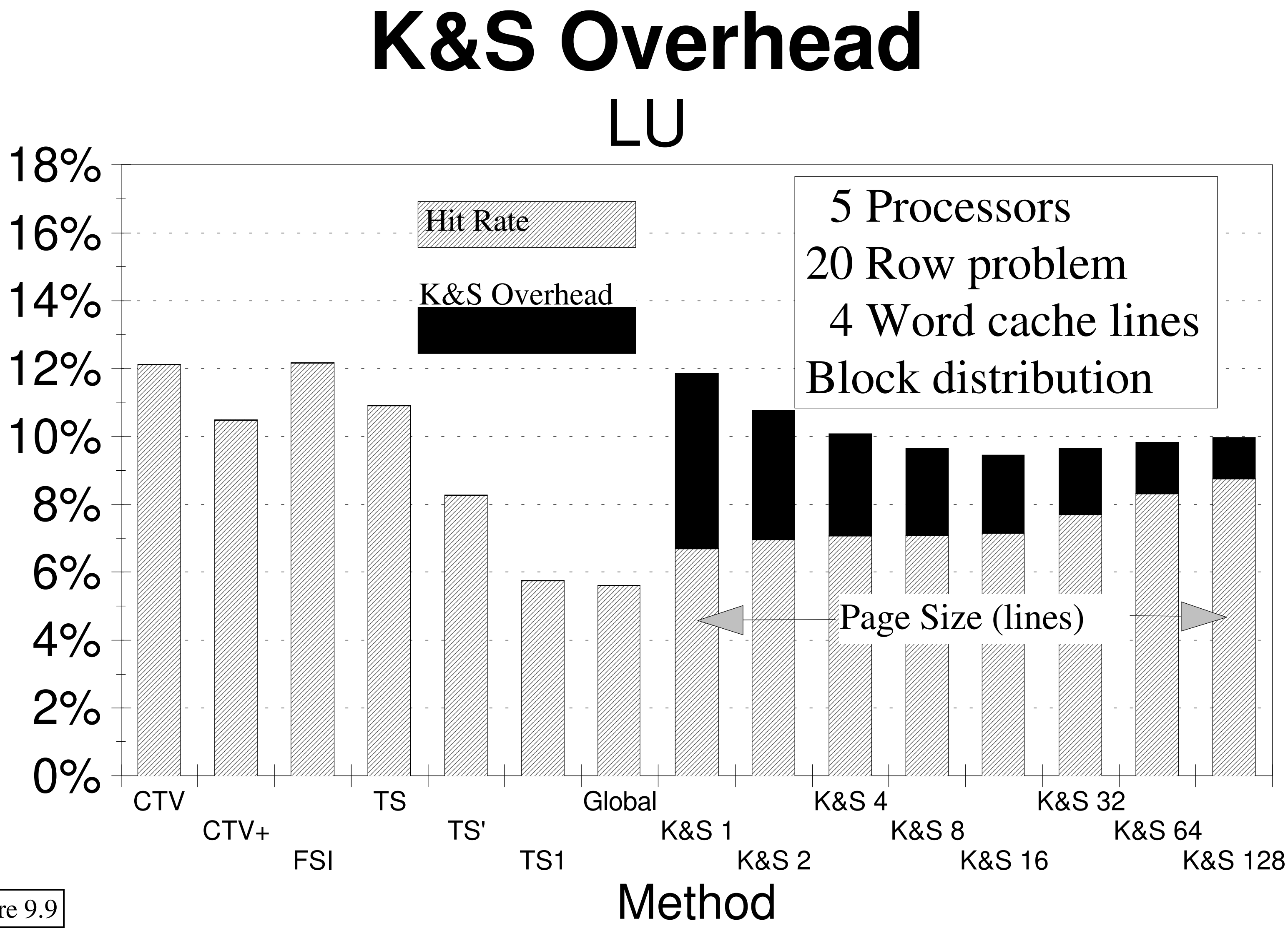

K&S Overhead
Erlebacher
14%
12%
10%
8%
6%
4%
2%
0%
Hit Rate
K&S Overhead
5 Processors
20 Row problem
4 Word cache lines
Block distribution
Page Size (lines)
CTV
CTV+
FSI
TS
TS'
TS1
Global
K&S 1
K&S 2
K&S 4
K&S 8
K&S 16
K&S 32
K&S 64
K&S 128
Method

Figure 9.8



K&S Overhead
LU
18%
16%
14%
12%
10%
8%
6%
4%
2%
0%
Hit Rate
K&S Overhead
5 Processors
20 Row problem
4 Word cache lines
Block distribution
Page Size (lines)
CTV
CTV+
FSI
TS
TS'
TS1
Global
K&S 1
K&S 2
K&S 4
K&S 8
K&S 16
K&S 32
K&S 64
K&S 128
Method

Figure 9.9

Scaling

Heat Flow, blocked, 4-word cache lines

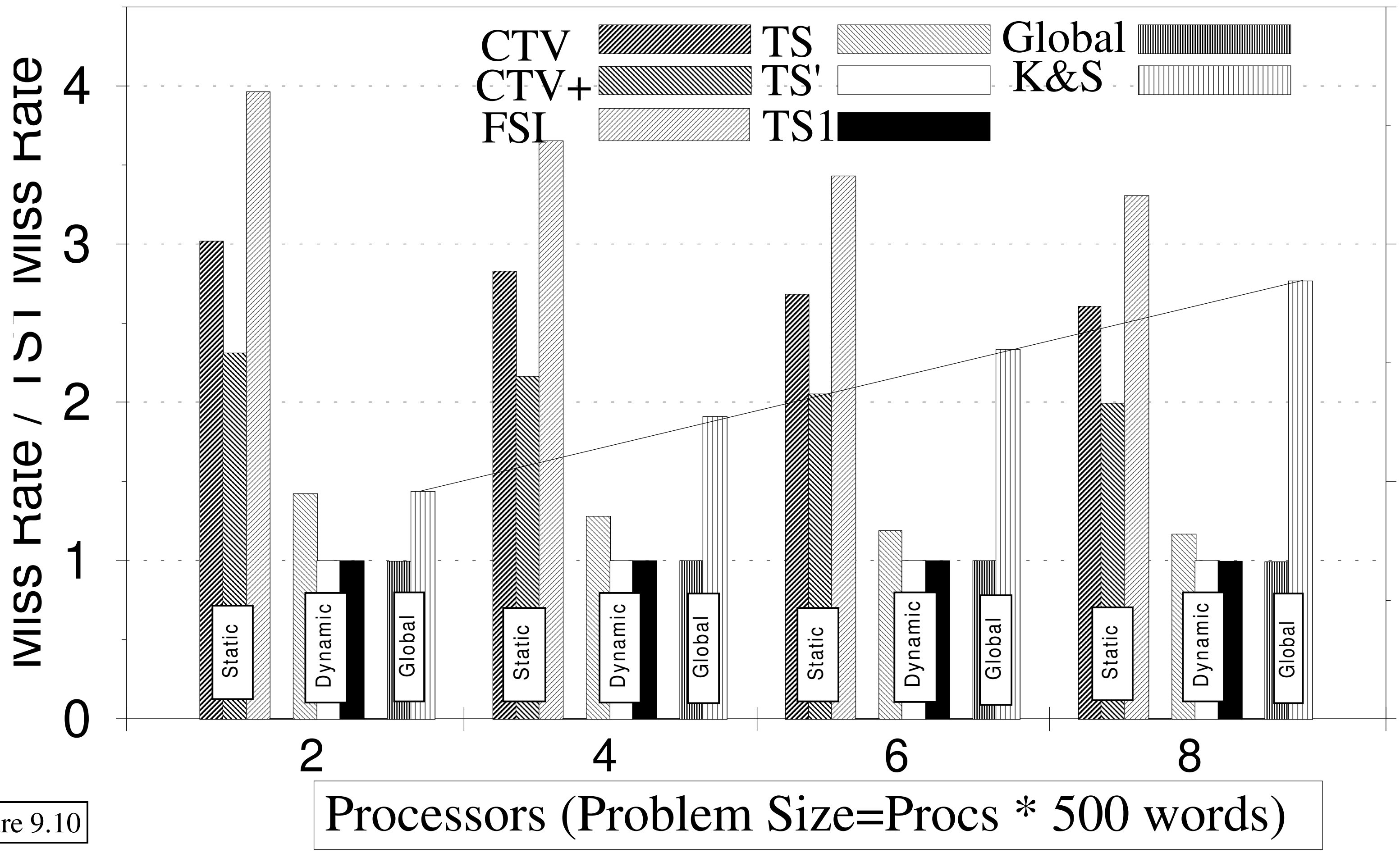


Figure 9.10

Scaling

Heat Flow, blocked, 4-word cache lines

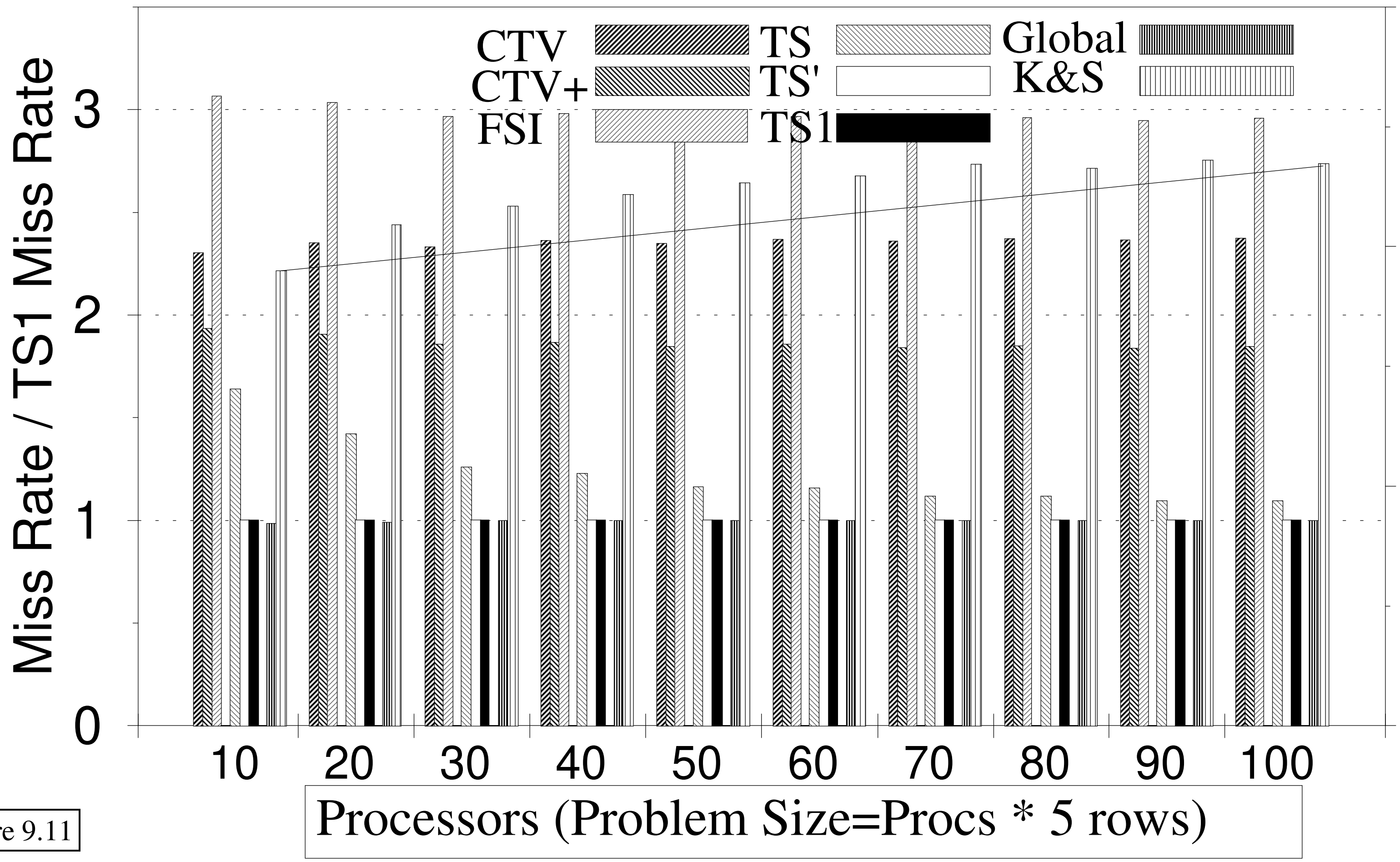


Figure 9.11

9.5 Significance of Data

The effectiveness of a coherence strategy is determined by the actual execution time (and implementation cost), not the miss ratio. However, collecting actual execution times in the simulator would depend on making several timing assumptions. Among them are:

1. Coherence overhead cost on a miss (for global strategies)
2. Per reference overhead for implementing a coherence strategy (including on hits)
3. Overhead for executing coherence instructions

All of these could vary depending on the actual implementation. Choosing any particular set of parameters might produce an answer for a given hypothetical machine, but it would obscure the more fundamental relationship of the strategies which is represented in the miss-rate data. Final conclusions can be reached by supplying whatever parameters seem appropriate for the above assumptions plus the miss-rate data.

We offer only the following qualitative observations about these parameters. Per reference overhead (cost (2)) is designed to be zero for the local strategies. That is, when an access hits in the cache, the coherence strategy does its work (if any) in parallel with returning the data and does not slow down normal accesses. A weak implementation of TS' would not preserve this property. This cost will be zero for many global strategies as well. It is non-zero for K&S, but can be considered as extra miss cost (§9.4.6).

The coherence cost (cost (1)) of a miss will be higher for global strategies than local strategies because of scaling. It includes the time for performing the coherence itself plus any compromise that must be made in basic memory response times to support the global coherence protocol. The magnitude of this depends on the particular global strategy being used. Different global strategies have significantly different cost functions for a miss. Local strategies have no cost penalty for misses.

The overhead of coherence instructions (cost (3)) is minimal for FSI, LSS, TS, & TS'. It amounts only to instructions taking some small, finite number of cycles outside of loops. Even for small programs, the expected cost is negligible. Asymptotically, the cost approaches zero. Since local strategies are concerned with scaling to large problems, the coherence instruction overhead cost can be considered zero for these strategies. The cost for CTV, CTV+, & TS1 is significant and needs to be considered. We already examined the situation with respect to TS1 (§7.3.2). CTV and CTV+ are considered appropriate where hardware cost is more important than instruction overhead. Thus, for strategies for which one would want to compare execution times, cost(3)=0.

The relative execution time of any two given strategies is then a matter of comparing, using whatever assumptions seem appropriate:

- misses*(base miss cost + cost(1))

The base miss cost is the time for the hardware to return a value in the absence of hardware coherence and is the same (on a given machine) for all strategies, by definition. This is not a model for predicting execution times; the parameters are average values over a complete run and depend on many complex interactions. Rather, it says that the only two parameters needed for comparing two strategies are the miss rate and the coherence overhead cost per reference.

9.6 Summary

The relative miss rates achieved by different coherence strategies depends mostly on the characteristics of the program being run. Parameters such as the program size, the number of the processors, the size of the cache, the cache line size, the cache line associativity, and the scheduling algorithm do not affect the relative miss rates. They may affect the total number of inter-epoch misses available, but coherence strategies continue to operate nearly equally on those remaining misses.

The two most important characteristics of programs are:

1) How much inter-epoch reuse exists.
2) How analyzable their subscript references are.

We summarize the relationship of the miss rates (but neither the execution time nor the implementation cost) for the different coherence strategies in figure 9.12. The granularity of the coherence is a direct trade-off for the miss rate, within each class of coherence strategy. K&S trades-off granularity for miss rate directly by altering the page size (the higher overhead for handling smaller page sizes is not represented in this graph). For local strategies the trade off is buried in the level of compiler analysis they can use.

The heavy solid arrows are relationships which provably must always be true. Global strategies always have an optimal hit rate, which will be the same as ideal local strategies only for analyzable subscripts. TS1 is an ideal local strategy and will have a better hit rate than other local strategy (static or dynamic). The light solid arrows represent miss rate relations that were always true in our test suite and which we expect to be true in nearly every case. The extra resolution of section analysis allows TS' with 4 colors to do better than TS (with unlimited clocks). The granularity of TS' is less than TS1 because of needing to color at the essential access. The miss rate spans a range because 4 colors will be sufficient to make an ideal coloring in some cases and not others. TS is further right still because it operates at the whole array level. It barely slips out of the ideal zone because of clock overflow. The dashed arrows represent what we observed to usually be true and believe to be the expected case, but will often be reversed. For K&S, the relation is drawn for a page size that produces the lowest total cost instead of lowest miss rate. That forces its level of granularity high enough that it has a higher miss rate than an ideal local strategy. One relation not in the transitive closure (or obvious) is CTV+ versus LSS. Our tests go both ways, with the section analysis sometimes compensating for LSS being dynamic.

The other costs for the coherence strategies are summarized in table 9.3. The important scales to examine them on are:

- C:M – Cache/Memory overhead - extra bits per cache line/per main memory word (excludes a single valid bit which is assumed for all caches)
- I – Instruction overhead - extra bits per reference instruction
- H – Hardware overhead - extra gates for implementation of protocol,
  - 1 – little, easy to implement
  - 2 – medium, acceptable but non-trivial
  - 3 – substantial, needs careful consideration
  - 4 – large, probably of theoretical interest only
- Inv – Invalidate overhead - time required for invalidate
- R – Reference overhead - excess time required for coherence when reference occurs, not including contention
- S – Scalability - run time cost of scaling processors and problem size. This measures contention and instruction overhead. These three are lumped together as "Inv:R/S" since a given method generally only suffers in one of them.
- G – Granularity of analysis
  - global – effective granularity is one word, no analysis is used
  - section – best possible at compile time, essentially array section
  - array – whole array
  - program – whole program
- 'Idealness' – How close to ideal does it come for a given granularity. The levels are:
  - optimal – no unnecessary hits occur (granularity irrelevant)
  - ideal – No local strategy could do better (worse than optimal)
  - static – achievable by a static strategy (worse than ideal)
- C – Effort that compiler must expend to implement method, same scale as 'H'
- Flow – The method handles inter-epoch control flow at run-time (§4.5.1)
- Post – Only post essential-write epoch analysis is necessary (§8.2.1)

The parameters are:

p	Number of processors
m	Size of main memory
n	Number of address bits, = $\log_2(m)$
s	Number of objects in an invalidate section

	C:M	I	H	Inv:R/S	G	Ideal	C	Flow	Post
Snoopy	0-1	0	1	:O(p)	global	optimal	0	yes	yes
Directory[3]	:O(p*m)	0	2	:O(n)[4]	global	optimal	0	yes	yes
SCI[36]	2*log(p) : log(p)[1]	0	2	:O(p)[2]	global	optimal	0	yes	yes
CKM	0	0	1	> O(s)	section	<< static	2	n/a	no
FSI	1	1	2	O(1)	program	< static	1	n/a	n/a
LSS	2	2[6]	3	O(1)	program	<< ideal	2	no	no
PEI	1	log n	4	O(1)[7]	< section	< ideal	2	yes	yes
TS	8?[8]	4+[9]	3	O(1)	array	ideal	1	yes	no
CTV	0	0	1	≈ O(s)	section	< static	2	n/a	no
CTV+	0	0	1	O(s)	section	static	2	no	no
TS1	1	0	4	O(log s)[5]	section	ideal	2	yes	yes
TS'	4	2	3	O(1)	section	< ideal	3	some	no

Table 9.3: Cost Comparison of Coherence Strategies

[1] Each cache entry needs two processor pointers and each memory block needs one.

[2] Worst case, expected case is much better.

[3] There are numerous directory-based coherence strategies that make interesting trade-offs between C/M and R. This entry describes a generic one. In particular, the actual bit cost is often p*m/16 or even k*m (for small constant k) with broadcast occasionally showing up as part of the R cost.

[4] This reflects increasing network depth to access the directory. Heavy sharing of a particular directory entry may cause some contention costs here (beyond actual costs of accessing the word).

[5] Hypothetical, we previously discussed the trade-offs (chapter 7).

[6] LSS has four kinds of reads, but only the one, usual, kind of write.

[7] Assuming an essentially fully associative cache and special power-of-2 layouts. PEI is not even usable if these assumptions cannot be met.

[8] Number of time stamp bits, typically 2-16, depending on how frequently clock overflow can be tolerated.

[9] $\log_2$(number of objects to track)

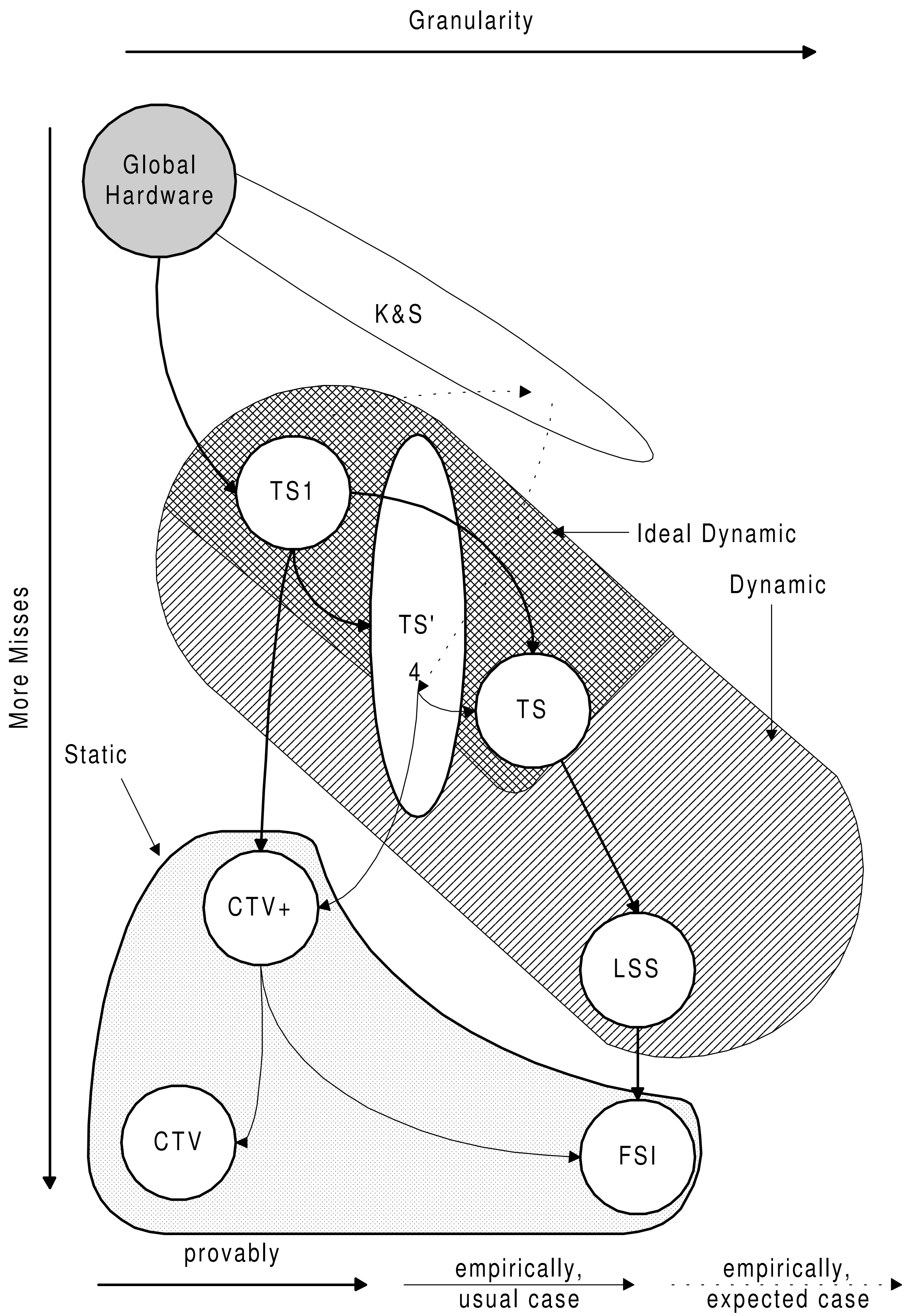


Figure 9.12: Miss Rate Relations

Chapter 10

Conclusions

Caching is an important technique for handling memory latency. It is especially critical for multiprocessors where memory delays can be much longer. Previous caching strategies were usually special hardware that kept track of every copy of a value (or found every copy by broadcast). VM strategies that use page faults to trigger coherence routines are another common method. The difficulty with these approaches is that they are not scalable because the cost of coherence increases as more processors are applied to larger problem instances.

We analyzed these approaches and previous compiler-assisted approaches at a fundamental level in order to unify their essential algorithms and establish their theoretical performance limits. Most of this analysis was performed without reference to any particular hardware implementation. Our conclusions will become increasingly important if future generations of machines continue the current trend of improvements in processor speed outpacing improvements in memory speed. We experimentally examined hit rates as a component of final performance that is invariant across hardware characteristics.

In this chapter, we review the most important insights from our study. In section 10.1, we review our framework for understanding coherence strategies. In section 10.2, we review the contributions of our three new approaches. In section 10.3, we summarize the application of coherence strategies by the most important determinant of their behavior, the class of program to which they are applied. In section 10.4, we speculate on the place of caching and coherence strategies in the larger picture of techniques for addressing latency and in future machines.

10.1 Knowledge Sources (Framework)

One of the contributions of this work is to recognize that there are two distinct sources of information that coherence algorithms need. How they are used determines the limits of the coherence algorithm. The two information sources are:

- The program dependences
- The schedule (of iterations to processors)

Global coherence strategies determine both the dependences and the schedule at run time; i.e., they have global information about a running program. With that complete information they can be *optimal*: All of the misses are logically necessary to move data (or because of cache size limits). The coherence protocol does not cause any additional misses. *Local coherence strategies* determine the dependence pattern at compile time and do not use the complete run-time schedule. All the information they use is local to a given processor. *Static* local strategies also estimate the schedule at compile time. *Dynamic* local strategies determine the local schedule at run time.

The schedule primarily refers to which processors get which iterations of a parallel loop. But it also refers to which branches are taken at execution time. Static strategies must make invalidate decisions based on every execution path whereas dynamic strategies can take advantage of the actual execution path. Nonetheless, static strategies can still capture much inter-epoch reuse by exploiting that fact that three epochs are required to make a cache line stale (for data-race free programs). For a broad, and well defined, set of program topologies we proved that a simple greedy algorithm preserves as much reuse as possible for a static strategy.

For dynamic strategies, the relevance to scalability is captured in the local schedule theorem (thm 3.5): the dependence pattern plus local scheduling information captures all of the useful information for purposes of coherence. There is no advantage in knowing the schedule on other processors. The useful information can be reduced to recognizing that a cache line is in one of three states: *fresh* (just referenced), *valid*, or *stale*. Fresh data can remain valid in the next epoch, regardless of the dependence analysis. This abstractly prescribes how local coherence strategies must work: local references at run time establish that part of the schedule is local and those references can be marked fresh and left in cache to generate inter-epoch reuse. This overrides compile-time estimates that such a value might have been written on a different processor. This allows local strategies to be scalable by not requiring any global knowledge at run time, not even how many other processors are sharing the work. Encoding the states and making the transitions can be done in a variety of ways. We demonstrated that all local strategies are nonetheless some flavor of this basic approach.

When the compiler can accurately analyze which values are written in a given loop, that information plus the local schedule is adequate in principle for a dynamic local strategy to be optimal, i.e. it will result in the same miss rate as a global strategy. Any schedule can be used, including load-balancing ones. In practice, a coherence algorithm may not be able to preserve all of the reuse that it can recognize (e.g. because the implementation would be too expensive). A dynamic local coherence algorithm that preserves all recognizable reuse is *ideal*. Its limiting factor is the accuracy of the compiler analysis and not the coherence algorithm *per se*. Ideal strategies fail to be optimal only when a value is used in one epoch, appears to be written in a subsequent epoch but is not actually referenced, then is reused in a 3^{rd} epoch.

Previous local strategies were presented only for one-word cache lines, where the physical and logical cache line are the same size. Local strategies must work on logical one-word cache lines to be useful. That does not prevent them from using longer physical cache lines to effectively prefetch and lower the miss rate. We established that the above three states must be recorded for each word (observation 8.17). This is in addition to its other control bits. We extended TS' to multi-word cache lines based on this observation and showed how previous local strategies could also be extended.

10.2 New Strategies

We proposed three new coherence strategies, each appropriate for different levels of hardware support.

CTV+ is a static local strategy. It was designed to work on machines which offer no support for coherence except the bare minimum of an invalidate. CTV+ is an alternative to not caching data, which would be the other option on such a machine. It succeeds in preserving approximately half of the available inter-epoch reuse.

TS1 was designed to have the best possible hit rate of any local strategy, even though it probably has no cost-effective hardware implementation. It is an *ideal* local strategy. It demonstrates that such can exist and are not mere abstractions. Its miss rate is an important measure of two things:

- The gap from an optimal global strategy that any local strategy must have.
- How much any given local strategy falls short of ideal and how much room there is for improvement.

TS' is a nearly ideal local strategy that could plausibly be used in a real machine. Its design objectives were that its hardware requirements be similar to those needed by previous local strategies and that it have low execution overhead. In particular, TS' uses a fast O(1) invalidate that requires it to compromise on being ideal. In the following discussion of where local strategies are effective, we have TS' in mind as the candidate strategy.

10.3 Different classes of programs

The relationships of the different coherence strategies on the test suite has already been discussed (§9.4) and summarized (§9.6). The significance of cache parameters was discussed in the same sections and found to be little. The relevant factor was found to be the data reference patterns in the program. In this section we summarize by that behavior instead of by the coherence strategy. These types apply separately to each recognizable array section, regardless of how they might be mixed in a particular program (e.g. if a program has one regular array and one irregular array, its performance for any given strategy will be a linear combination of the independent performance on each of the arrays). However, we discuss them by program where the main data array in a given program shows the particular behavior.

10.3.1 Analyzable Programs

By analyzable programs we mean those for which the compiler can exactly tell which locations are written in a given parallel loop. Ideal local strategies behave optimally (have the same miss rate as global strategies) on analyzable programs. On the reduction parts of LU, on all of Erlebacher, and all of Heat Flow, TS' with 4 colors was an ideal strategy. TS1 was ideal on all of the programs in our test suite except the row swap in LU. TS1 and TS' both did better than TS for this class of programs because they could utilize the better resolution of section analysis that is available because the program is analyzable.

For run-time costs, we partially explored TS1 versus TS (§7.3). Other than that, we regard TS1 as a thought experiment and do not attempt to compare its run-time costs to other strategies.

For TS', we can make firmer predictions. Since its hardware implementation is comparable to that of TS, we expect its run-time performance relative to TS to be a function only of the miss rate. For analyzable programs, it does significantly better in many cases and almost never worse (§9.4.4). For the intended implementation of TS', with no execution time overhead, it also does better than K&S or any global strategy, by virtue of being optimal, on analyzable programs for which it has sufficient colors; i.e., its miss rate is the same and its overheads lower. In our test suite, we found four colors to be sufficient for unit cache lines and five colors to be sufficient for non-unit cache lines. We expect that to be true for computational kernels in general (where most reuse is to be found). Regardless of how many colors are needed, the only interesting implementation of TS' is one with sufficient colors to handle most programs. If the economic tradeoffs make sense to implement it at all, then it should be faster than a global strategy.

10.3.2 Regular, but not analyzable

These are programs in which the data movement follows a regular pattern and has a high degree of sharing yet appears to be completely irregular to compiler analysis. We previously gave a contrived, extreme example (§9.4.6). This is the worst case for local coherence strategies where performance drops to that of TS, but still there is significant reuse to be gained. None of the programs in our test suite behaved this way. If such programs are common, local strategies will never be adequate.

10.3.3 Irregular

In this class of programs there is little inter-epoch reuse because the data moves between processors each epoch. The movement may be regular from an algorithmic point of view, but it is not from a scheduling or compiler analysis point of view—FFT is an example. There is some small incidental reuse because the data may randomly end up on the same processor. TS' and TS capture this random reuse only if it spans two adjacent epochs. TS1 did capture nearly all of it in the case of FFT (including that which spanned multiple epochs) and would also do so for any loop for which the write set can be expressed symbolically in closed form at the end of the loop. This gap between TS' and TS1 suggests that there is still some room for improvement in local strategies. The extra dependence analysis resolution of TS' gives it no advantage over TS for this class of programs since that analysis is not likely to return any useful information. That TS' performs no better in this case is not a great loss since there is little to be gained, even for a global strategy.

Only in unusual cases can global strategies use their extra flexibility to good advantage. The miss-rate gap between TS' and a global strategy will be small (§9.4.5):

- For a large number of processors
- When most of the data in the irregular section is written each epoch
- When most of the possible hits are intra-epoch

Considering run-time costs, TS' did better than K&S on FFT because the overhead for the latter was much greater than the extra hits gained. We expect this to be true in general for irregular programs. There is much data movement to track because cache lines keep moving. At the same time, there is little actual sharing to be preserve. That is a worst case scenario for K&S. Any global strategy will face a similar difficulty though it could have substantially lower overheads than K&S. Whether or not TS' is a viable alternative to a given global strategy will depend on the details of those tradeoffs.

10.3.4 Partially analyzable

Partially analyzable programs are those for which some accesses are not analyzable. But other references to the same data are analyzable. This class of programs presents special challenges for local strategies separate from simply being a combination of the two types. LU is an example of this. Most of the references are analyzable and local strategies are effective on them. However, the row swap actually resembles an irregular computation. Preserving the reuse on the irregular part is not so important since there is not much to preserve and it is only a fraction of the references. What is important is that it not compromise the miss rate for the regular part of the data. As with FFT, TS1 was able to preserve nearly all of this reuse (a smarter TS1 able to symbolically evaluate a conditional preserved it all), but TS' lost much of it because it needed to know the color at the essential access. TS' was still able to use the more refined section analysis to do substantially better than TS. Also, TS' did better than K&S which suffered substantial overheads because of constantly shifting block distributions. A global strategy with low overhead costs would perform better than TS' on this class of programs.

10.4 Perspective

In a broader view than just comparing coherence strategies, no one approach to reducing latency will be the most suitable one for all classes of programs. From this investigation into the types of programs for which local coherence strategies perform well and for which they perform poorly, we have gained some perspective, though no firm prescription, on the applicability of the different approaches. This is summarized in figure 10.1.

The horizontal dimension represents how thoroughly the compiler can analyze the dependence pattern. Moving right means that the strategy simply does not try for high resolution subscript analysis (e.g. TS) and it also means that it tries for the best possible analysis but cannot analyze the subscript. The vertical dimension represents how much of the schedule (i.e. data placement) the compiler knows. For programs where the analysis is incomplete, it also represents how much inter-epoch reuse there really is (how irregular the program is even for a perfectly known schedule).

The right edge represents those references for which global hardware coherence is obligatory. Synchronization primitives are in this class. They must act in a sequentially consistent fashion. The compiler can obviously not be used to predict such real time events. Such references can be handled on a special basis (e.g. by skipping cache) and do not imply a need for global coherence hardware for all of cache.

The next most erratic kind of reference (from a compiler point of view) is unrestricted pointer use. The compiler might not be able to analyze the pointer reference and may even find the rest of the data unanalyzable. In such a situation any compile-time decision about where to invalidate or where to place data will probably be wrong. The full flexibility of a global strategy is needed.

Returning to the left edge of the figure, to programs that are completely analyzable, compile-time techniques are ideal here. If the schedule is also known, SPMD is the right approach because it has complete information and can place data near where it will be referenced at run time. If the schedule is unknown, e.g. dynamic load balancing, a local strategy is the right approach because it moves data closer, into local cache, while preserving all available reuse. An SPMD approach would be trapped into making many unnecessary remote references.

If the underlying regularity of the computation stays the same, but the analysis becomes coarser, a local strategy is favored because an initial placement becomes more difficult. For instance, if the compiler can reasonably predict that all of an array is written (e.g. an indirection array), but not which parts are written where, a local strategy will still achieve an optimal hit rate, but a static layout would not have had sufficient information to produce a good layout. Therefore, the border between local strategies and SPMD runs toward the upper right of the diagram.

Distributed shared memory models (DSMs[4]) are like the SPMD model in the sense that the addresses for a given distributed array will not be contiguous, the compiler must make layout decisions, and coherence is handled implicitly in the structuring of the computation. The important difference is that a DSM works with request and reply, so the owner does not need to know what to send. When data is known to be remote, this is a disadvantage because of the delay involved in the request and lack of opportunity to bundle messages. But, when there is good chance that what might be a remote reference is actually local, the DSM has an advantage because there will be no useless pre-loop on the receiver and no extra code at all on the sender. The original layout still needs to be reasonably accurate, DSMs are not caching data (at the coherence level). For this reason, load balanced scheduling does not favor DSMs. They do have an advantage over SPMD for programs with some small number of difficult subscripts and essentially regular accesses.

When there is insufficient information to make a good initial placement decision, global strategies can do a better job. When the actual computation is more irregular, global strategies are also favored. The border between DSMs and global strategies runs to the upper right, as does the border between SPMD and local strategies. Ultimately, the reasons are the same, the degree of actual regularity times the percent analyzability of it needs to cross some threshold for a static placement strategy to be effective.

The situation for local coherence versus VM strategies is different because the tradeoff is not about how good the compile-time information is but instead it is about run-time overhead costs. For the same level of analysis, increasing irregularity hurts VM strategies because of the extra overhead of tracking moving copies, but it does not impact local strategies. In this context, we view an increasingly sparse computation (q in §9.4.5) not as a more irregular one (because the data movement still looks ‘random’) but as a less analyzable one because compile-time analysis less accurately predicted the total amount of data moved. This is consistent with the border between local and VM strategies running to the lower right.

---

[4] “DSM” is also used to refer to VM-based coherence systems (e.g. LRC) where the distribution is physical and the sharing is handled on a software basis. These operate like cache with replication and coherence issues. We use DSM in a different sense here.

That decreasingly regular and increasingly load-balanced programs favor local strategies should not come as a surprise. An objective in their design is that scheduling information cannot be used, therefore less predictable scheduling removes the advantage that other strategies would have in trying to use it. Local strategies have already made the worst case assumptions about scheduling.

10.4.1 Future Directions

The trend in supercomputer design is to build hardware DSM systems that physically have distributed memory but logically support global addresses by special hardware, e.g. the Cray T3D [19] and CM-5 [34]. Such machines support message passing *and* transparent addresses at a primitive level. They do not support hardware coherence. Instead, shared data must be left uncached (but hopefully local for most references) or it must be handled via a VM-style coherence strategy. More efficient caching would help performance, but is viewed as too expensive to implement.

This is exactly the environment for which a DSM approach to latency is appropriate: data is laid out in local memory but can also be fetched remotely with simple references whenever needed. When the extra information is available, SPMD approaches should be more efficient and can also be used by being built on top of the message passing primitives. Finally, VM approaches can be used, as they can in almost any machine. All of these approaches can be mixed on the existing hardware platforms.

Local strategies are also part of this picture. While no current machine supports the hardware necessary for a local strategy, it could be incorporated into a DSM hardware structure. Local strategies can be used in conjunction with other approaches. For example, local strategies can be applied as an optimization to a DSM approach. They can be applied directly at the cache level beneath the local memory level which is managed by the DSM. If the local strategy determines that a value can be left in cache across an epoch boundary, an otherwise remote reference can be avoided. Local strategies can also be applied as the sole coherence mechanism, to either the cache proper (beneath local memory) or the local memory itself where its addresses are being remapped as necessary (e.g. COMA).

The challenge is to tailor a program such that different types of data access are each handled by the most appropriate mechanism. The complimentary problem is to design machines that are efficient for the approach to latency most often used on problems intended for the machine. The advantages of each approach can be realized and the disadvantages minimized. Local strategies are strongest where other approaches are weakest and offer improvement for an important part of the problem domain.

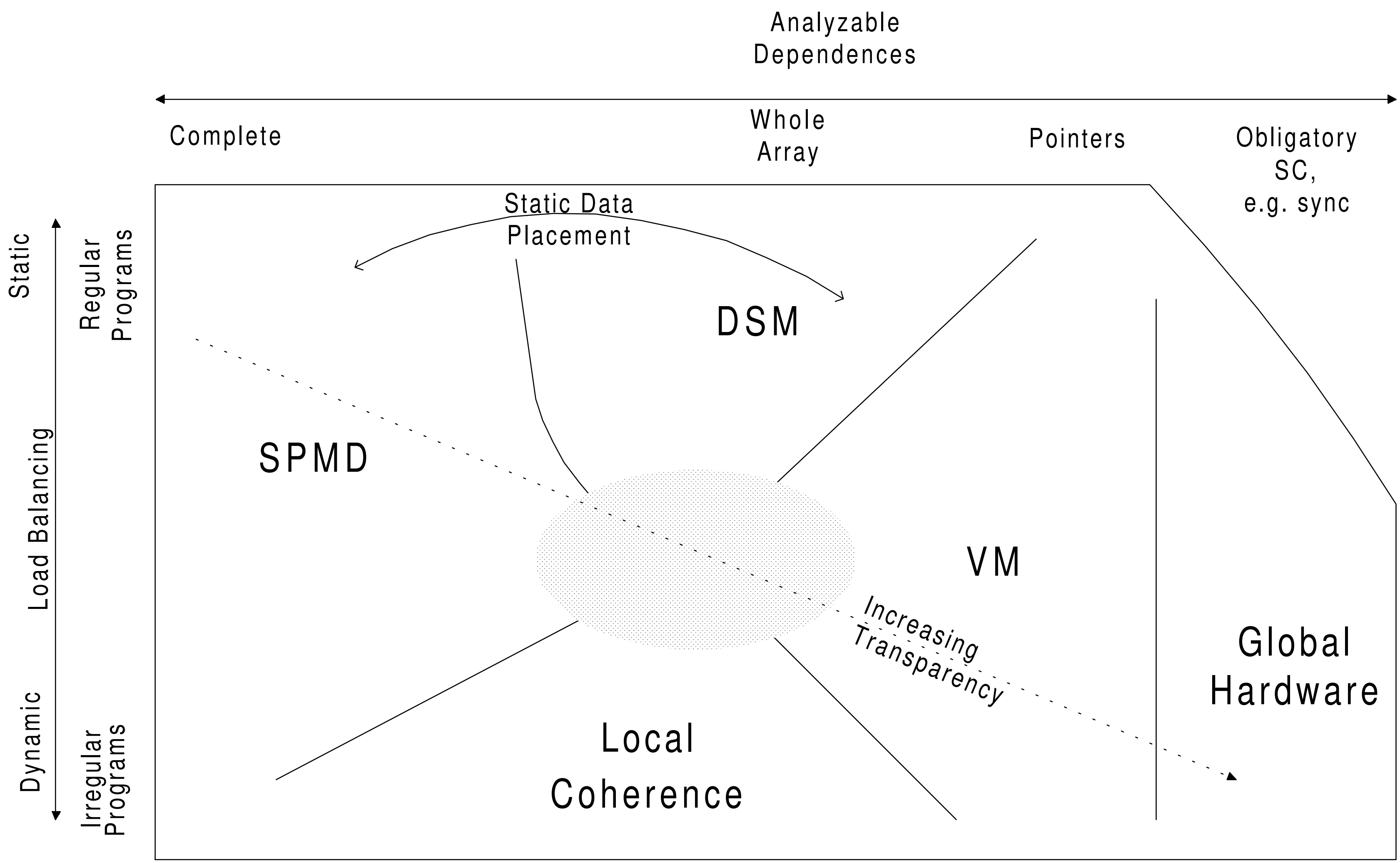


Figure 10.1: Perspective on Approaches

Bibliography

[1] D. A. Abramson, K. Ramamohanarao, and M. Ross. A scalable cache coherence mechanism using a selectively clearable cache memory. *The Australian Computer Journal*, 21(1), Feb. 1989.

[2] S. Adve and M. Hill. Unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613-624,June 1993.

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. *Proc. of the 15th Annual International Symposium on Computer Architecture*, 1988.

[4] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491-542, Oct. 1987.

[5] G. A. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting Heterogeneous Parallelism on a Multi-threaded Multiprocessor. *In Proceedings of 1992 International Conference on Supercomputing*, July 1992.

[6] J. Archibald. A Cache Coherence Approach For Large Multiprocessor Systems. *Proceedings of the 1988 International Conference on Supercomputing*. pages 337-345, July 1988.

[7] D. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. The NAS parallel benchmark Results 10-94. NAS Technical Report NAS-94-001, October 1994.

[8] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63-73, Fall 1991.

[9] François Bodin, Elana D. Granston, and Thierry Montaut. Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing. *Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994. Also available as CRPC Technical Report No. CRPC-TR94479 at `gopher://softlib.rice.edu/99/softlib/CRPC-TRs/reports/CRPC-TR94479.ps`.

[10] S. Carr. Memory-Hierarchy Management. PhD Thesis, Department of Computer Science, Rice University, September 1992.

[11] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112-1118, December 1978.

[12] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49-58, June 1990.

[13] H. Cheong. Life-span strategy - a compiler-based approach to cache coherence. *In Proceedings of 1992 International Conference on Supercomputing*, July 1992.

[14] H. Cheong and A. Veidenbaum. A Cache Coherence Scheme With Fast Selective Invalidation. *Proc. of the 15th Annual International Symposium on Computer Architecture*, 1988.

[15] H. Cheong and A. Veidenbaum. Compiler-directed cache management for multiprocessors. *Computer*, 23(6):39-47, June 1990.

[16] L. Choi and P-C Yew. Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. *Supercomputing* '94.

[17] Convex Computer Corporation. Convex Exemplar Architecture. Convex Press, Richardson, TX, November, 1993

[18] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. Efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31-58, 1991.

[19] Cray Research, Inc. CRAY T3D System Architecture Overview. 1993

[20] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[21] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. *Proc. of the 1988 International Conference on Parallel Processing*, pages 229-238, Aug. 1988.

[22] E. Darnell and K. Kennedy. Cache Coherence Using Local Knowledge. *Supercomputing* '93.

[23] E. Darnell, J. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. *Proceedings of 1992 International Conference on Supercomputing*, July 1992. Also available as expanded Technical Report CRPC-TR92197, Center for Research on Parallel Computation, January 1992.

[24] J. Dongarra J., et al. Solving linear systems on vector and shared memory computers. *Society for Industrial and Applied Mathematics*, Philadelphia, 1991.

[25] T.M. Eidson and G. Erlebacher, Implementation of a fully-balanced periodic tridiagonal solver on a parallel distributed memory architecture. Submitted to *Concurrency: Practice and Experience*, also available as an ICASE Report 94-17, NASA CR 194892.

[26] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, 1979.

[27] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proc. of the 18th Annual International Symposium on Computer Architecture*, 1990.

[29] G. Goff and K. Kennedy and C.-W. Tseng. Practical Dependence Testing. *SIGPLAN*, June 1991.

[30] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[31] J. Goodman and P. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. *Proc. of the 15th Annual International Symposium on Computer Architecture*, 1988.

[32] A. Gupta and W-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

[33] A. Gupta, W-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. *Proc. of the 1990 International Conference on Parallel Processing*, pages I312-321, August 1990.

[34] D Hillis and L W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31-40, November 1993.

[35] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66-80, August 1992.

[36] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Scalable coherent interface. *Computer*, 23(6), June 1990.

[37] N. P. Jouppi. The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance. Digital Corporation Western Research Laboratory Technical Research Report 89/13, July 1989. Also available as: `http://www.research.digital.com/wrl/publications/abstracts/89.13.html`

[38] S. Karlovsky. Automatic management of programmable caches: Algorithms and experience. Technical Report 89-8010, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, July 1989.

[39] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proc. of the 19th Annual International Symposium on Computer Architecture*, 1992.

[40] Kendall Square Research. KSR1 Technical Summary. Waltham, MA, 1992.

[41] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1993.

[42] L. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. *First International Symposium on High Performance Computer Architecture*, Raleigh, NC, January 1995.

[43] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[44] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63-79, Mar. 1992.

[45] K. Li. Comparative Evaluation of Cache Coherence Schemes Based on Virtual Memory Support. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[46] Lilja, David. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computer Surveys*, 25(3), Sept. 1993.

[47] A. Louri and H. Sung. A compiler directed cache coherence scheme with fast and parallel explicit invalidation. *Proc. of the 1992 International Conference on Parallel Processing*, pages 2-9, Aug. 1992.

[48] Nathaniel McIntosh. An Empirical Study of Inter-loop Reuse in the NAS benchmarks. Technical Report CRPC-TR95519, Center for Research on Parallel Computation, March 1995.

[49] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Proceedings of 1992 International Conference on Supercomputing*, pages 104-113, Nov. 1992.

[50] S. Min and J. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25-44, Jan. 1992.

[51] S. Min, J. Baer, and H. Kim. An efficient caching support for critical sections in large-scale shared-memory multiprocessors. In *Proc. of the 1990 International Conference on Supercomputing/Computer Architecture News*, pages 4-47, June 1990. Special issue of *Computer Architecture News*, 18(3), Sept. 1990.

[52] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of Runtime Support for Parallel Processors. *In Proceedings of 1988 International Conference on Supercomputing*, July 1988.

[53] Motorola, Inc. PowerPC 604 Data Sheet. Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona 85036. 1995. Also available as the web page: `http://www.chips.ibm.com/products/ppc/DataSheets/sheets.html`

[54] T. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching. PhD Thesis, Department of Electrical Engineering, Stanford University, March 1994.

[55] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Technical Publications, San Diego, CA, 1989.

[56] K. Peterson and K. Li. Cache coherence for shared memory multiprocessors based on virtual memory support. *Proceedings of the 7th International Parallel Processing Symposium*, Apr. 1993.

[57] G. Pfister and V. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *Proceedings of the 1985 International Conference on Parallel Processing*, pages 790-795, August 1985.

[58] D. Schanin. The design and development of a very high speed system bus - the encore multimax nanobus. *Proceedings of the Fall Joint Computer Conference*, pages 410-418, Nov. 1986.

[59] M. Tomasevic and V. Milutinovic. Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors. *IEEE Micro*, October 1994.

[60] R. Wilson. Content-addressable memory: the solution finally finds a problem. In *Computer Design*, page 105, Dec. 1988.

[61] J. Willis, A. Sanderson, and C. Hill. Cache coherence in systems with parallel communication channels & many processors. In *Supercomputing '90*, pages 554-563, 1990.

[62] M. E. Wolf. Improving Locality and Parallelism in Nested Loops. PhD thesis, Department of Computer Science, Stanford University, August 1992.

[63] R. Zucker. Relaxed Consistency and Synchronization in Parallel Processors. PhD thesis, Department of Computer Science and Engineering, University of Washington, December, 1992.

Appendix A

Notational Conventions

To reduce the size of examples and to remove statements not relevant to the issue at hand, several special notations are used in this thesis.

Epochs are numbered sequentially (generally in run-time order) and named e_1, e_2, etc. These refer to actual run-time epochs and not static parallel loop nests.

Code examples are always presented in a courier font, e.g. '`A(I)=I+1`'. Quotes are often omitted where the intent is clear. Meta-syntax in code examples is italicized, e.g. '`IF (`*c1*`)`' means that '*c1*' has some run-time significance that is unknown or irrelevant at compile-time. In particular, '`IF (`*false*`)`' means that at run-time the condition will be false but at compile-time this cannot be determined. '`A(`*f*`(I))`' means that `A` is indexed by some function or expression involving `I` that cannot be analyzed.

Comments in code which starts a separate line are included to the right in a Times-Roman font without any explicit syntax or punctuation, e.g.

```
A(I)=A(I)+1        A  is incremented
```

`DO` loops are compressed in various ways. In this work, it will often be the case that reference patterns inside of a loop are important but neither the actual loop bound nor the computation is. A loop might be abbreviated:

```
DO I
   A(I)=
   =B(I)
   C(I)=B(I)+1
   D(I)+=1
ENDDO
```

This iterator for the `I` loop is omitted as irrelevant. `A(I)` is assigned some value by some function not involving `A`. `B(I)` is used to compute some other value which is not relevant. `C(I)=B(I)+1` is a complete statement with no elision. `D(I)+=1` is a 'C'ism with the obvious meaning in complete Fortran code.

Often it will be useful to refer to different references to the same variable. These will be numbered with a superscript. This is merely annotation and not part of the variable name or the syntax. For example, the last line in the previous example might appear as:

$\mathtt{D}^1\mathtt{(I)=D}^2\mathtt{(I)+1}$

Similarly, when it is necessary to refer to the epoch a reference occurs in, it will be prefixed with the epoch name. Note that the epoch for a given reference may change while the reference superscript does not because the former is a dynamic notion and the latter a static notion (this matters principally for a PDO inside of a serial loop). For instance, the previous assignment when referenced in epoch 3, would appear as:

$e_3\mathtt{:D}^1\mathtt{(I)=D}^2\mathtt{(I)+1}$

Parallel loops will be represented by PDO. In some examples, there will be so many epochs that the PDO will be omitted also. The contents of a parallel loop are then squeezed into a single line so that the epoch boundaries are still clear. All left hand sides from the original loop are listed as a group on the left of the single assignment and similarly for the right hand sides. For example,

```
PDO I
   A(I)=B(I)+1
   C(I)=A(I)*2
ENDDO
```

would be compressed to:

```
A,C=A,B
```

Where used this will not sacrifice any accuracy because the intra-epoch flow will not be relevant and all apparent dependences will be the relevant inter-epoch ones. There should be no confusion with scalars because such highly compressed examples will not reference any scalars. Also, arrays are always represented by A, B, C, or D (or variables that start with these letters). Index variables are always represented by I, J, or K. Other scalars use any other letter as appropriate. Code examples lifted from real programs (instead of created for explanatory purposes) will be complete and not follow the abbreviation or naming conventions.

A_r	A is read
A_w	A is written
A_{rw}	A is read and written
A_{ac}	A is accessed (read or written)
A_l	A is loaded (in the cache line), but not accessed
$A_r[i]$	A is being read on p_i
$A_w[\neg i]$	A is being written on any processor except p_i.
$\neg A_{ac}$	A is not accessed

Table A.1: Reference Types Notation

When referring to references in isolation (not in a code example) and the actual subscript is not relevant, they will be subscripted with the type of reference (table A.1). These can summarize whole epochs or refer to single references.

Processors (and their local caches) will be referred to as p_0, p_1, ...p_i etc.. Where it is useful to discuss particular schedules, processors will be referred to as in Karlovsky [38], by a processor number in brackets following references.

Appendix B

Test Suite

Our test suite consisted of four scientific Fortran programs:

> Heat Flow – a 2-D relaxation code
> LU Decomposition – a blocked right-looking Gaussian elimination [24]
> Erlebacher – a tridiagonal derivative solver [25]
> FFT – a 2-D FFT

These represent distinct styles of programs with respect to coherence issues. They stress the weaknesses of different strategies.

B.1 Heat Flow

Heat Flow is a simple 2-D relaxation. Its core code is:

```
DO T
   PDO I
      DO J
         Grid1(x,y)=r*(Grid2(x-1,y)+Grid2(x+1,y)+
                        Grid2(x,y+1)+Grid2(x,y-1))
                   + r41*Grid2(x,y)
      END DO
   END DO
   PDO I
      DO J
         Grid2(x,y)=r*(Grid1(x-1,y)+Grid1(x+1,y)+
                        Grid1(x,y+1)+Grid1(x,y-1))
                   + r41*Grid1(x,y)
      END DO
   END DO
```

The sharing pattern is simple for this code. It is an easy case for coherence since all of the subscripts are known before the time step (`T`) loop, the sharing pattern is regular, and high reuse is expected. The interesting aspect is preserving reuse for the border which is only read and not written. The border is simply the single row and column along the outside:

```
Grid1(1,1:N),Grid1(N,1:N),Grid1(1:N,1), Grid1(1:N,N)
```

`Grid2` is similar and the rest of both array is the ‘interior’. For purposes of coherence, the code reduces to essentially:

```
DO t
   Grid1.interior = f(Grid2.interior ,Grid2.border)
   Grid2.interior = f(Grid1.interior ,Grid1.border)
```

B.2 LU Decomposition

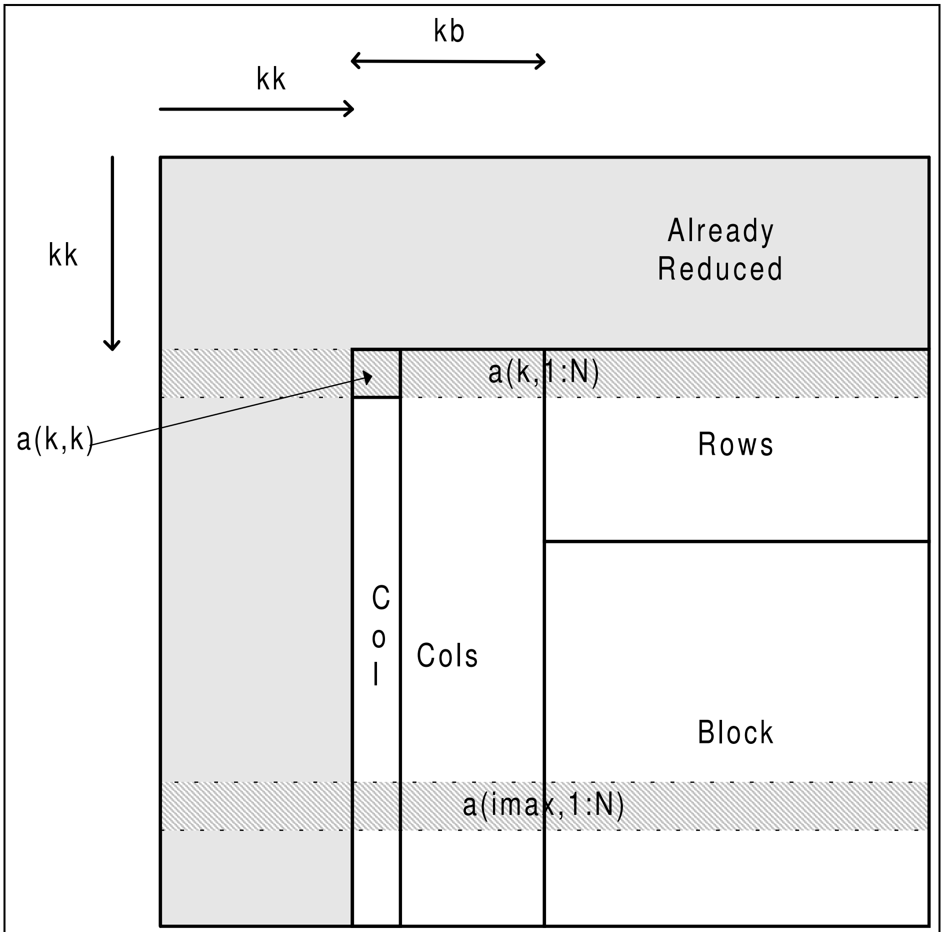


Figure B.1: LU Decomposition

LU Decomposition is a blocked right-looking Gaussian elimination. Essentially, it is a partially pivoted Gaussian elimination, but a block of pivots is computed each time instead of a single one. The reduction step then reduces the remaining sub-matrix across as many rows as are in the block. The block size was 5 in our tests. Note that this does not in anyway constrain the available parallelism. All of shared data and work occurs in a single array.

The matrix as it appears in the middle of a reduction is shown in figure B.1. The block size is `kb`. The algorithm in outline form is:

```
for each block (kk)
   for each k in block
      find a pivot & swap rows
      reduce column under pivot ('Col')
      reduce other columns in block ('Cols')
   end
   reduce 'Rows' with all pivots in parallel
   reduce 'block' with all pivots in parallel
end
```

This challenges a coherence strategy because the sharing involved in the '`block`' reduction changes for every iteration of the block (`kk`) loop. Still, there are important pieces that can be shared between each of the reduction steps when the sections involved are understood. The second challenge for a coherence strategy is the search for a pivot is completely opaque to the compiler. Any row could be written during the swap. In summary, LU is essentially a regular computation but with a mix of analyzable and unanalyzable references. The essential part of the code is:

```
integer kb – block size
integer kk – starting row of current block
FORK
do kk=1,m-1,kb
   do k=kk, kk+kb-1
      JOIN
      imax is such that ∀i, a(imax,k) ≥ a(i,k)
      if (imax.ne.k) then
         swap a(imax,1:n),a(k,1:n)
      end if

      // Reduce 'Col'
      a(k+1:m,k) /= a(k,k)
      FORK

      // Reduce 'Cols'
      PDOB j=k+1, kk+kb-1
         a(k+1:m,j) -= a(k+1:m,k)*a(k+1:m,j)
      end do
      BARRIER
   end do
   BARRIER

   // Reduce 'Rows'
   PDOB j = kk+kb,n
      do k = kk, kk+kb-1
         a(k+1:kk+kb-1,j) -= a(k+1:kk+kb-1,k)*a(k,j)
      end do
   end do
   BARRIER

   // Reduce 'Block'
   PDOB j=kk+kb,n
      do i=kk+kb,m
         a(i,j) -= a(i,kk:kk+kb-1)*a(kk:kk+kb-1,j)
      end do
   end do
   BARRIER
end do
```

B.3 Erlebacher

Erlebacher calculates derivatives over a tridiagonal system. The main computation involves seven three dimensional arrays and a variety of different boundary conditions each of which are handled in separate loops. That makes the computation heavily (loop) distributed with lots of epochs. The challenge for a coherence strategy to achieve good performance on Erlebacher is recovering reuse which spans several epochs . The computation is easily analyzable with all of the subscripts known early. The core computation spans most of the code, but most of the aspects of it that are relevant to coherence can be explained with reference to the following fragment:

```
PDO 10 k=1,N
   f(1,1:N,k) *= b(1)
end do
BARRIER

PDO 20 k=1,N
   do j=1,N
      f(2:N-1,j,k)=
         (f(2:N-1,j,k)-a(2:N-1)*f(1:N-2,j,k))*b(2:N-1)
   end do
end do

PDO 30 k=1,N
   tot(1:N,k) = 0.0
end do
BARRIER

PDO 40 k=1,N
   do 40 j=1,N
      tot(j,k) += d(1:N-1)*f(1:N-1,j,k)
   end do
end do
BARRIER

PDO 50 k=1,N
   f(n,1:N,k) -= tot(1:N,k))*b(n)
end do

PDO 60 k=1,N
   f(n-1,1:N,k) -= e(n-1)*f(n,1:N,k)
end do
BARRIER

PDO 70 k=1,N
   do j=1,N
      do i=N-2,1,-1
         f(i,j,k) -= c(i)*f(i+1,j,k) - e(i)*f(n,j,k)
      end do
   end do
end do
```

B.4 FFT

This is a simple 2-D FFT. The subscript reference is linearized into a single subscript and it changes in a complex fashion. FFT is essentially unanalyzable, primarily because the upper bound of the `J` loop, `nx`, changes in a non-linear fashion. The actual computation is also irregular (at least from a compiler point of view). There is little room for a coherence strategy to achieve much with FFT. The challenge is only to keep overheads low. This is in contrast to LU which is partially unanalyzable but more regular in the actual computation. The core of FFT is:

```
incrm2 = m
incrm = m/2
nx = 2
depth = Log2(m)
do k = depth-1, 0, -1
   PDO j=0,nx/2-1
      tmp(j*incrm2+incrm) = ...
      fac(j*incrm2+incrm) =
         f(tmp(j*incrm2+incrm),tmp(j*incrm2+incrm))
      do i=j*incrm2+incrm, j*incrm2+2*incrm-1
         x(i) = x(i-incrm) - x(i)*fac(j*incrm2+incrm)
         x(i-incrm) += term2(i)
      end do
   end do
   nx = nx*2
   incrm2 = incrm
   incrm = incrm/2
end do
```

Appendix C

K&S

K&S works for general acquire and release synchronization, but is presented here for only the simple barrier case. The pseudo-code is this author's interpretation of the K&S algorithm and is written in C++ in order that the objects emphasize where the actions take place (local to the processor or in the main memory).

The key to the efficiency of K&S is that every processor locally maintains a *weaklist*. *Weaklist* contains every dirty shared page that is in use by the local processor. At a barrier, a processor then simply scans its local *weaklist* and invalidates every page it finds there. The amount of work done is proportional to the amount of local sharing instead of total shared pages and nearly all of the work is done locally. The precise state is maintained in global main memory, but most of that work happens in conjunction with the actual cache-miss induced access.

The expensive part of the algorithm is maintaining *weaklist*. Updating it requires extra accesses to other processor's local memories that are not necessary for the computation itself (the critical step is marked in bold in code below). As a compromise, *weaklist* does not exactly reflect which pages are dirty and shared. Instead, it promises only to contain all dirty shared pages; it may contain others as well. When a page first becomes dirty and shared, *weaklist*s are updated. But pages are not removed from *weaklist* until there are no processors sharing the page. In general, it may be the case that a page is read-only shared but still appear in a *weaklist*. However, in the case of simple barriers only, every page on every *weaklist* is invalidated at the same time at the barriers. Therefore, *weaklist* contains only pages which actually need to be invalidated (unless they have already been evicted).

The algorithm is:

```
typedef int Tpid;       // processor number
typedef int Tpage;      // memory page
// as stored in Main Memory
class MainMemoryPage {
   enum {uncached,shared,dirty,weak} state;
   int count;           // # of sharing processors
   SetOfTpid Map;       // Processors currently using this page
} Pages[size of memory];
```

```
// Stored locally to each processor
class Processor {
   Tpid pid;
   void Barrier ();
   SetOfTpage WeakList;
   // Add i to set of weak pages on this proc
   MakeWeak(i) {WeakList += i;}
} Procs [number of processors];

// Invalidate all pages written by other proccessors during this epoch
CProcessor::Barrier ()
{
   for (Tpage i∈WeakList) {
      Invalidate(i);      // physically invalidate page on this processor
      Pages[i].Map -= pid;// Remove from set of procs caching page
      if (--Pages[i].count ==0 ) state=uncached;
   }
   Weaklist=∅;
}

// If new list of sharing processors mandates weak state,
// inform those sharing processors.
void CMainMemoryPage::CheckWeak (bool Write) {
   if (count>1 && (Write || state==dirty)) {
      if (state!=weak) {        // just became weak
         state=weak;
         for (Tpid p∈map) Procs[p].MakeWeak(i);
      }
   }
}

// Add p to list of processors sharing page i
// And update processor local data structures if needed
void CMainMemoryPage::Add (Tpid p, bool Write) {
   Map += p;    // Add p to set of procs caching page i
   Count++;
   CheckWeak(Write);
   if (state!=weak) state = (Write) ? dirty : shared;
}

// Invoked on an access fault
// Write is true if the access was a write
void AccessFault (Tpid p, TPage i, bool Write)
{
   // Is this a write fault on an already mapped, but
   // read-only page?
   if (p∈Pages[i].Map)
```

```
        Pages[i].CheckWeak(Write);
    else    // Or, is it a fault to an unmapped page?
        Pages[i].Add(p,Write);
}
```

Appendix D

SPMD (Single Program, Multiple Data) Examples

SPMD[35] adds explicit communications to a parallel program such that a program written for a shared address space is transformed into one that refers to the local address space for each processor. The transformation guarantees correctness and thus removes the need for coherence, values are sent and received instead of being shared. The advantages and disadvantages are discussed as part of our survey of other approaches to latency (§2.1.2). Here we include some examples as further explanation

The best case for SPMD is when the compiler can fully analyze the program and no communication is needed. For example, an SPMD approach applied to the following code fragment [35]:

Consider the following trivial loop:

```
DO I=1,100
   A(I)=B(I)+1
ENDDO
```

If there were 4 processors working on the computation, the work could simply be distributed in a block fashion. Processor 0 handles iterations 1 through 25, etc. In that case, `A(25*p+1:25*(p+1))` is accessed only on processor `p`. `A(75:100)` is allocated in memory local to processor 3 and referenced locally as `A(1:25)`, etc. In this case, there are no network communication costs. The code would be transformed as follows and executed on every processor independently:

```
DO I=1,10
   A(I)=B(I)+1
ENDDO
```

Communication will usually be necessary. Exactly what to communicate needs to be calculated as does the local index function. Consider the following loop, with the same distribution of work on 4 processors, as before:

```
REAL A(100)
DO I=1,95
   A(I)=A(I+5)
ENDDO
```

It needs to be determined if a reference is local or remote. In this case, `A(26)` is local to processor 1 and remote to processor 0, but processor 0 needs it when it writes `A(21)` (assuming the owner-computes rule). Processor 1 needs to explicitly send `A(26)` and processor 0 needs to know it is remote and to explicitly receive it. Its address will be different on both processors and have no obvious relation to the address of `A(26)`. The final result for a block distribution on 4 processors is:

```
REAL A(30)
my$p = myproc()       – 0 ≤ my$p ≤ 3
if (my$p.gt.0) send(A(1:5),my$p-1)
if (my$p.lt.3) recv(A(26:30),my$p+1)
ub$1=min((my$p+1)*25,95)-(my$p*25)
do i=1,ub$1
   A(i)=A(i+5)
end do
```

Since the address space is local, the compiler must know with certainty which processor owns the data needed by a given reference. When the subscript is not completely analyzable, the compiler will need to insert code to run-time evaluate the subscript and convert it to the processor owning the data. With shared memory systems, this process is directly handled in hardware by routing the request to the correct memory bank based on the address, a much lower overhead process. For instance, if the assignment in a loop were:

```
do i=1,100
   B(i)=A(f(i))
end do
```

An SPMD approach would generate code such as (again for 4 processors and a block distribution):

```
do i=1,100
   i$1=f(i)
   own$p=i$1/4
   if (own$p.eq.my$p) then send (owner(B(i)),A(i$1))
   if (my$p.eq.owner(B(i)) then
      recv (own$p,A(i$1))
      B(i)=A(i$1)
   end if
end do
```

owner(`B(i)`) can be calculated at compile-time. But each processor must execute the full 100 iterations to calculate `own$p` in order to decide if it needs to send `A(i$1)`. This destroys some of the benefit of the parallelism by forcing each processor to consider every iteration instead of just its share. With respect to `A`, no locality is preserved. Every value must be communicated. Nothing is saved over faulting and fetching the value. It is even a little bit worse because if `A(i$1)` already resides on the same processor as `B(i)`, there is still the overhead of checking.