Issues in Instruction Scheduling

Philip Schielke
Rice University
Department of Computer Science

Ph.D. Thesis Proposal

Abstract

Instruction scheduling is a code reordering transformation that attempts to hide latencies present in modern day microprocessors. Current applications of these microprocessors and the microprocessors themselves present new parameters under which the scheduler must operate. For example, some multiple functional unit processors have partitioned register sets. In some applications, increasing the static size of a program may not be an acceptable tradeoff for gaining improved running time. The interaction between the scheduler and the register allocator can also dramatically affect the performance of the compiled code.

In this thesis we will look at global scheduling techniques that do not replicate code, including scheduling over extended basic blocks. We also look at a replacement to the traditional list scheduler based on the techniques of iterative repair. Finally, we explore the interaction between instruction scheduling and register allocation, and look at ways of combining the two problems.

1 Introduction

Instruction scheduling is one of the last phases performed by modern compilers. It is a code reordering transformation that attempts to hide the latencies inherent in modern day microprocessors. On processors that support instruction level parallelism, it may be possible to hide the latency of some high-latency operations by moving other operations into the "gaps" in the schedule. When compiling for machines that have hardware interlocks, the instruction scheduling phase may not be necessary for correctness, but is performed in the hopes of improving the code. On machines that do not have hardware interlocks (*i.e.* NOPs have to be inserted into the code to correctly enforce dependences between operations) instruction scheduling must be performed.

Throughout this work we will have in mind a VLIW machine model. (Although, we feel the work applies also to superscalar architectures). That is, we assume a processor with multiple pipelined functional units on which several operations can be initiated at the same time. The compiler assigns each operation to a functional unit, and assigns a start time to the operation. In light of this, it will help to make clear a couple of definitions. We will distinguish machine level *operations* from *instructions*. An *operation* is a single indivisible command to the microprocessor (*e.g.* an add operation or a load operation). An *instruction* is a collection of operations that are initiated on the same machine cycle. Each operation in a given instruction executes on a different functional unit.

The concept of instruction scheduling grew out of the microcode community. A good review of microcode compaction can be found in Landskov, *et. al.* [15]. In order to make microcodes as small and fast as possible, the microcodes are "compacted". Originally this was done by hand, but eventually algorithms were developed to do this automatically. At first algorithms for instruction scheduling were purely local. That is, each basic block was scheduled independently. Unfortunately, most basic blocks that have been heavily optimized by a good optimizing compiler are small, and present little opportunity for finding instruction level parallelism [20]. Even at the basic block level, finding an optimal solution to all but the simplest scheduling problems has been shown to be NP-complete [7].

In this work we will explore several aspects of instruction scheduling as they relate to modern processors and processor applications. We are interested in scheduling techniques that reduce static code size (in addition to execution time), and methods to combine instruction scheduling and register allocation. Section 2 presents an overview of list scheduling – the classic algorithm for instruction scheduling. The rest of the work explores methods that go beyond mere list scheduling over basic blocks. Section 3 looks at three shortcomings of list scheduling over basic blocks and introduces our methods for overcoming them. section 4 explores techniques to perform global instruction scheduling, focusing on methods that require no code replication. Section 5 looks at alternative techniques to list scheduling, focusing on repair based methods. Section 6 explores the relationship between register allocation and instruction scheduling.

There is one other approach to scheduling that must be mentioned, namely cyclic scheduling, or *software pipelining*. Software pipelining is a scheduling technique that applies only to loops. The gaps in the schedule of the loop are filled with operations from other iterations of the same loop. Lam describes software pipelining in some detail [14]. Software pipelining often uses a list scheduler as one of its phases. Thus the techniques described in Section 5 may be applicable to one aspect of software pipelining. However that application is beyond the scope of this work.

2 List Scheduling

The traditional way to schedule a basic block (and as we'll see later, larger program regions) is to use *list scheduling* [10, 15, 7]. The first step, is to build a data precedence graph (DPG) (*aka* data dependence graph or DDG) for the block. The nodes of the graph are operations in the basic block, and the edges represent dependences between operations. An edge from node A to node B, means that node A must complete its execution before node B can begin. We will call the nodes in the DPG that are not dependent on other operations (*i.e.* have no incoming edges), *leaves*. Likewise, nodes with no outgoing edges we will refer to as *roots*.

In the next step, priorities are assigned to each node in the graph. There are several different heuristics that can be used to assign priorities. A common and effective strategy is to use the latency weighted depth of the node [10, 15]. The *depth* of a node n is the length (number of nodes), of the longest path in the DPG from n to some root (including n and the root.) The *latency weighted depth* is computed the same way, but the nodes along the path are weighted using the latency of the operation the node represents. The following formula summarizes the priority computation for a node n.

$$priority(n) = \max\left(\forall_{r \in roots(DPG)} \forall_{p \in paths(n,\ldots,r)} \sum_{p_i = n}^{r} latency(p_i)\right)$$

Dynamic programming can be used to compute the priorities efficiently:

$$priority(n) = \begin{cases} latency(n) & \text{if } n \text{ is a root.} \\ latency(n) + max_{(m,n) \in E}(priority(m)) & \text{otherwise.} \end{cases}$$

The final phase is the actual list scheduling algorithm that constructs the schedule for the block. Starting at cycle 0, the list scheduler places operations into the schedule cycle by cycle. Any operation that is "ready" at cycle X (*i.e.* all its operands have been computed), is a candidate to be scheduled at cycle X. The priorities computed in the previous step are used to determine which of the ready operations to schedule, by selecting the highest priority operation first. Any tie in the priority of two operations is broken arbitrarily. The algorithm is detailed in Figure 1.

Instruction scheduling under any set of realistic assumptions has been shown to be NP-complete [25]. List scheduling is a polynomial time algorithm that selects one schedule out of the space of possible solutions. Assuming the DPG is built, (an $O(N^2)$ computation), we examine the complexity of the list scheduler as presented in Figure 1. Consider line (1) of the algorithm. Each operation is removed from the ready list exactly once. If a heap based priority queue is used as the ready list, this part of the code contributes $O(N \log N)$ to the running time. If we assume a constant bound on the latency of an operation, line (2) contributes $O(N)$. In line (3) we traverse each outgoing edge of the node. This is done for all nodes, giving a complexity of $O(E)$. Thus the worst case is when $E = N^2$, giving a worst case complexity of $O(N^2)$. For a typical scheduling problem $E = O(N)$ and the number of ready operations can be bound by a constant. Thus, list scheduling is typically linear in practice once the DPG has been built.

Input: Data Precedence Graph (N,E) with priorities assigned to each node. Parameters of machine (instruction latencies, pipelining, number of functional units, etc.)

Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine.

Algorithm:

```
cycle = 0
ready-list = leaf nodes in DPG
inflight-list = empty list
while ( ready-list or inflight-list not empty, and an issue slot is available )
(1)     for op = (all nodes in ready-list in descending priority order)
            if (a functional unit exists for op to start at cycle)
                remove op from ready-list and add to inflight-list
                add op to schedule at time cycle
            endif
        endfor
        cycle = cycle + 1
(2)     for op = (all nodes in inflight-list)
            if (op finishes at time cycle)
                remove op from inflight-list
(3)             check nodes waiting for op in DPG and add to ready-list if all operands available
            endif
        endfor
endwhile
```

Figure 1 List Scheduling algorithm

3 Shortcomings of Basic Block List Scheduling

In the previous section we looked at the traditional way to perform instruction scheduling, list scheduling over basic blocks. This section looks at some of the deficiencies of that technique and introduces our approaches to overcoming them.

3.1 The scope of scheduling

The first shortcoming of list scheduling over basic blocks is the scope of the technique. Basic blocks tend to be small and offer a limited amount of instruction level parallelism. To address this problem, various approaches for regional or global scheduling have been proposed over the years. It is the goal of these techniques to find more parallelism by scheduling several basic blocks together. As an example consider the assembly-like code in figure 2. We assume a processor with two identical functional units. Assume that `LOAD` and `MULT` require two cycles to complete, and all other operations require only one cycle. The unscheduled code for two basic blocks is show in part (a). In part (b) each basic block has been individually list scheduled. In part (c) we have moved some operations between blocks, to simulate what a non-local technique may be able to do. Of course we must ensure that these motions are safe, but the example shows the potential benefit of a non-local scheduling technique. We have reduced the total execution time for these two blocks from 11 cycles in part (b) to nine cycles in part (c).

Unfortunately many global scheduling techniques increase the size of the object code to gain runtime improvements. In the next section we will look briefly at some of these techniques. Very little work has been done on non-local scheduling with the intent of improving the runtime performance of the object code *and* keeping the object code small. (One exception to this is Sweany's dominator-path scheduling, also discussed in the next section.) We endeavor to answer the questions, "Are significant runtime speedups possible without increasing the code size?" and "Can non-local instruction scheduling actually reduce static code size?" In the next section we formulate extended basic block scheduling. This is a relatively simple approach to regional instruction scheduling that requires no code replication, and only a small amount of data-flow analysis.

3.2 The greediness of list scheduling

A second shortcoming of basic block list scheduling, is the list scheduling algorithm itself. As presented, the algorithm greedily constructs a single schedule from the solution space of all possible schedules based on a given priority heuristic. This schedule may or may not be the shortest possible schedule. In addition, we may seek other schedule qualities (such as reduced register pressure, or improved dynamic execution properties) not easily incorporated into a list scheduling heuristic. At the same time, performing an exhaustive search on the space of all schedules may not be feasible for a given problem instance. In section 5 we look at some examples where list scheduling fails to find the optimal solution and we explore some alternatives to list scheduling, including randomized backward and forward list scheduling and iterative repair.

As we will see, iterative repair offers several advantages over list scheduling. First, it uses a stochastic method (guided by heuristics) to generate a schedule in the solution space. Thus, many schedules can be generated and the shortest schedule (or "best" schedule by some other criteria) is kept. The iterative repair algorithm works by making small modifications to a complete (but not always correct) schedule. Therefore, it is able to see the outcome of a scheduling choice before committing to it. This added flexibility comes with the cost of longer scheduling times.

3.3 Interaction between scheduling and allocation

The final shortcoming we will address, is the tendency of list scheduling to increase the live ranges of registers, thus making register allocation more difficult. We could perform register allocation before scheduling, but that would introduce artificial dependences into the DPG making scheduling more difficult. Since it is not clear that we want to perform one of these phases before the other, some sort of combined scheduling/allocation

```
foo: LOAD @a    => r0
     LOAD @b    => r1
     ADD  r0 r1 => r2
     LOAD @c    => r3
     MULT r3 r0 => r4
     SUB  r4 r2 => r5
     STOR r5 @d
     COMP r4 0  => r6
     BREQ r6 bar baz
bar: LDI  0 => r7
     STOR r7 @e
     EXIT
```

(a) Unscheduled code

```
foo: LOAD @a    => r0 | LOAD @b   => r1
     LOAD @c    => r3 | NOP
     ADD  r0 r1 => r2 | NOP
     MULT r3 r0 => r4 | NOP
     NOP              | NOP
     SUB  r4 r2 => r5 | COMP r4 0 => r6
     STOR r5 @d       | NOP
     BREQ r6 bar baz
bar: LDI  0 => r7     | NOP
     STOR r7 @e       | NOP
     EXIT
```

(b) Basic Block Scheduling

```
foo: LOAD @a    => r0 | LOAD @b   => r1
     LOAD @c    => r3 | LDI  0    => r7
     ADD  r0 r1 => r2 | STOR r7 @e
     MULT r3 r0 => r4 | NOP
     NOP              | NOP
     SUB  r4 r2 => r5 | COMP r4 0 => r6
     BREQ r6 bar baz
bar: STOR r5 @d       | NOP
     EXIT
```

(c) Global Scheduling

Figure 2 Example showing benefits of Non-local Scheduling

phase looks appealing. In section 6 we explore the tension between scheduling and register allocation in more depth, and give a detailed example of the problem. We plan on combining allocation and scheduling under the iterative repair framework.

4 Effective Instruction Scheduling with Minimal Code Growth

4.1 Some global scheduling techniques

As stated earlier, basic blocks offer only a limited amount of instruction level parallelism. Because of this, several "global" scheduling techniques have been developed over the years in an attempt to discover more of the instruction level parallelism available in the program.

There are several approaches that have been used to perform global instruction scheduling. At first, *ad hoc* methods were used to move operations between basic blocks that had already been scheduled in the hopes of improving the overall schedule [9]. *Trace Scheduling* was one of the first automated global scheduling techniques. In trace scheduling, the most frequently executed (loop free) path through the function is determined using profile information. This "trace" is treated like a large basic block. A DPG is created for the trace, and the trace is scheduled using a list scheduler. All the restrictions on moving operations between basic blocks are encoded in the DPG. After the first trace is scheduled, the next most frequently executed path is scheduled, and so on. A certain amount of "bookkeeping" must be done when scheduling the trace. Any operation that moves above a join point in the trace must be copied into all other traces that enter the current trace at that point. Likewise, any operation that moves below a branch point must be copied into the other traces that exit the branch point, if the operation computes any values that are live in that trace. Fisher describes trace scheduling in some detail [9].

Hwu, *et.al.* present another global scheduling technique know as *superblock* scheduling [13]. Superblocks are similar to traces except that during the construction certain blocks are duplicated to remove all join points in the traces. First, traces are built in like fashion to trace scheduling. Second, *tail duplication* is performed to remove side entrances into the trace. During tail duplication, all the blocks between the first side entrance and the end of the trace are replicated. Then, all other side entrances into the trace are redirected into the copy. This duplication can then be repeated on the duplicate. Therefore, a superblock can have several branch points, but only one entry point. This method reduces the amount of book keeping required, and making several copies of certain basic blocks can lead to greater opportunities for finding instruction level parallelism.

These methods present a major change in the thinking behind instruction scheduling; code can be replicated in order to improve the overall execution time of the program. Unfortunately, the tradeoff for improved runtime efficiency has been growth in the static size of the executable code. In many (most) computing environments this tradeoff has been acceptable. Memory and disk space are fairly cheap, and most users are willing to sacrifice some of those resources for improved runtime performance.

There are, however, some cases where static code growth is undesirable. The work in instruction scheduling first stemmed from a desire to make microcodes as small as possible, since these microcodes would be burned into the ROM's of the finished computer. As instruction scheduling found application on modern day microproccesors, the issue of small executables became less important. But today we see complex microprocessors and DSPs being used in commodity applications such as microwaves, cellular phones, and other embedded systems. Since the code for these applications is typically burned into a ROM, small code size results in a smaller ROM, saving both money and power. The focus on small static code size has come full circle.

4.2 Controlling static code size

Both trace scheduling and superblock scheduling increase the size of the resultant code. Although this code growth can be controlled to a certain degree, there is little information on how effective these global scheduling methods can be when code growth is restricted.

Dominator-path scheduling is a scheduling technique that does not require any code replication. Sweany and others have done work in this area [21, 23, 12]. A dominator path is any path in the dominator tree of a function. The nodes in a dominator path (basic blocks) have the property that if one block executes, its parent in the dominator path must also have executed. This scheduling method prohibits moving an operation between basic blocks, if that move would require the operation to be copied to another basic block. Bernstein, *et. al.*, use control dependence information (via the Program Dependence Graph or PDG) to guide the scheduling process [2]. A set of nodes with the same control dependences are scheduled together. This set of nodes has the interesting property that if one of the nodes execute, all of them do.

In this work we will be looking at various scheduling techniques. We would like to evaluate these techniques in both a local and a global instruction scheduling paradigm. We are also concerned with keeping static code sizes small. We have developed an alternative global scheduling technique called Extended Basic Block (or EBB) Scheduling. This technique is similar to Superblock scheduling, except that we don't allow blocks to be duplicated. In essence every block that represents a join point in the control flow graph (CFG), becomes the first block in some extended basic block. Therefore, like superblock scheduling, the regions of code over which we schedule, have no join points.

An EBB is a sequence of basic blocks $B_1, \ldots, B_k$ such that for $1 \leq i < k$, B_i is the only predecessor of B_{i+1} in the control-flow graph (CFG), and B_1 may or may not have a unique predecessor [1]. For scheduling purposes we view extended basic blocks as a partitioning of the CFG; a basic block is a member of one and only one EBB. The EBB selection algorithm is outlined in Figure 3. First the EBB "header" blocks are selected and placed on the work-list. Initially, header nodes are all blocks with more than one predecessor in the CFG, and the start block. The EBB's are built by taking a header off the work-list and adding blocks to the EBB based on size. When the algorithm reaches a block with more than one successor it needs to select one of the blocks to be in the current EBB. The other blocks go onto the work-list as new headers. This choice is made by selecting the successor with the highest EBB-*priority*. Before building EBB's, we compute the EBB-priority for each basic block. The heuristic we use for the EBB-priority is based on the size (number of operations) of the basic block. The EBB-priority for a block with multiple predecessors (*i. e.* the initial set of header blocks) is simply the number of operations in the block. The EBB-priority for all other blocks is equal to the number of operations in the block plus the maximum EBB-priority of all the block's successors in the CFG. The EBB-priority for all blocks is computed in a bottom up walk over the CFG.

Once the EBB's are computed, a DPG must be built for each EBB. The construction is identical to the construction for basic blocks, except that additional edges must be inserted to prevent certain operations from moving between basic blocks. Let $B1$ be a basic block and $B2$ and $B3$ be the immediate successors of $B1$ in the CFG. Also, let $B1$ and $B2$ be members of the same EBB, and let $B3$ be in some other EBB. This is shown in figure 4 We prohibit an operation from moving from $B2$ "up" to $B1$ if the operation computes a value that is live along the edge from $B1$ to $B3$. Similarly, we prohibit an operation from moving down from $B1$ to $B2$ if the operation computes a value that is live along the edge from $B1$ to $B3$. In addition certain operations that may cause exceptions are not allowed to move up in the control flow graph. In the following experiments, no divide operations are allowed to move up, and certain loads are also prevented from moving up. In addition to choosing EBB's based on size, we restrict all basic blocks in an EBB to be at the same loop nesting depth.

We have run some experiments that compare list scheduling over basic blocks to list scheduling over EBB's. We use a hypothetical VLIW-like machine model. Our machine has five functional units: two integer units, one floating point unit, one memory unit, and one branch unit. Up to four operations can be initiated each cycle, and all functional units are completely pipelined. In addition, we have assigned various latencies to each operation.

Table 1 shows the results for the simple VLIW model. We compare extended basic block scheduling to basic block scheduling. List scheduling with the latency depth priority heuristic is used as the underlying scheduling algorithm. We assume that the code passed to our hypothetical VLIW must include NOP's to ensure that dependences are enforced. The column labeled "Static Insts" is the total number of instructions in the program including NOP instructions. The "Dynamic Insts" column is the dynamic instruction count and can be thought of as an estimate of the number of cycles it would take to run the program.

We see quite a range of reductions due to EBB scheduling. This is due to different amounts of instruction level parallelism available in the programs, and the amount that we were able to exploit using EBB scheduling.

Input: Control Flow Graph (N, E), with each block annotated with an EBB priority.

Output: A set of Extended Basic Blocks that partitions the CFG.

Algorithm:

for *node* = (all nodes in CFG)
    if (*node* has more than one predecessor in CFG) **or**
        (*node* is the start block)
        then add *node* to *work-list*; mark *node* as a header
    endif
endfor
while (*work-list* is not empty)
    select a node n from *work-list*
    current = n
    do
        max = the successor of *current* with the largest EBB priority that is not a header
        add *max* to n's EBB
        for *succ* = (all successors of *current*)
            if *max* $\neq$ *succ* **and** *succ* is not a header
                then add *succ* to *work-list*
            endif
        endfor
        current = *max*
    while (*current* is not nul)
endwhile

Figure 3 Creating Extended Basic Blocks

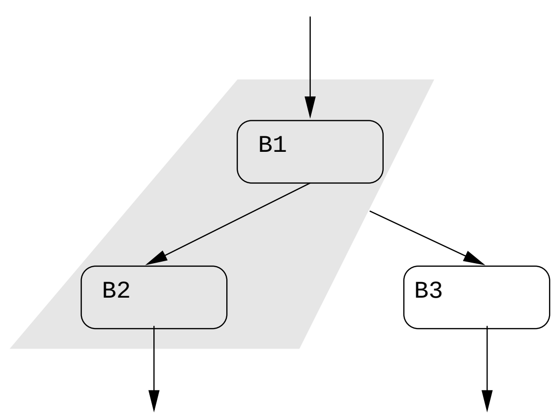


Figure 4 Example CFG. Shaded area is one EBB

Note also that reductions in static code size do not always lead to a similar reduction in execution time. If most of the code size reduction comes from compacting infrequently executed portions of the code, that reduction does not translate into improved runtime behavior. Contrariwise if we shorten the code on a frequently executed path, we could see a larger reduction in the run time of the code.

There are several ways to potentially improve EBB scheduling. The first is how the EBB's are built. The size heuristic we described above may not be the best. We intend to experiment with some other EBB selection heuristics. In addition, we currently restrict all basic blocks in an EBB to be at the same loop nesting depth. This could be relaxed in the hope that some code may be moved out of loops. The second possible improvement is in the underlying scheduler. The latency depth heuristic may not be the best list scheduling heuristic we could use. Since we are scheduling over EBB's, it may be profitable to modify the heuristic to push code "down" in the CFG as much as possible. Finally, we can look at scheduling over multiple EBB's. That is, instead of viewing EBB's as a partitioning of the blocks in the program, we schedule over all possible EBB's, or some subset of them. Liveness information will have to be recomputed or incrementally updated during the scheduling process.

We plan to implement a dominator path scheduler and compare the results of the two techniques, as well as compare the techniques themselves.

	Basic Block Scheduling		Extended Basic Block Scheduling			
Benchmark	Static Insts	Dynamic Insts	Static Insts	% decrease	Dynamic Insts	% decrease
`clean`	11429	4515619	9973	12.7	3769537	16.5
`compress`	1597	10641037	1376	13.8	9625986	9.5
`dfa`	1356	696450	1000	26.3	594859	14.6
`fft`	2700	22469970	2513	6.9	22150570	1.4
`go`	73456	589209782	54002	26.5	501223760	14.9
`jpeg`	19785	45900780	17480	11.7	42529297	7.3
`nsieve`	271	2288889385	249	8.1	2254242028	1.5
`water`	6469	36111497	6039	6.7	33651055	6.8
`fmin`	712	5370	503	29.4	4495	16.3
`rkf45`	2387	818884	2055	13.9	761549	7.0
`seval`	1055	3340	993	5.9	3283	1.7
`solve`	1007	2813	935	7.1	2671	5.0
`svd`	2481	14649	2234	10.0	13963	4.7
`urand`	192	1117	172	10.4	1081	3.2
`zeroin`	545	4603	446	18.2	4095	11.0
`applu`	13334	884028559	12933	3.0	866954001	1.9
`doduc`	42127	16953587	38536	8.5	16141047	4.8
`fpppp`	10521	95701038	9798	6.9	90647430	5.2
`matrix300`	429	43073238	361	15.9	42883523	0.4
`tomcatv`	953	436717483	912	4.3	436707058	0.0

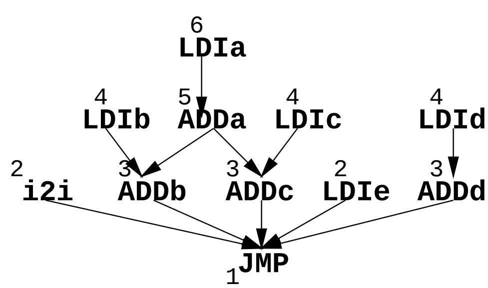
Table 1 EBB Scheduling for simple VLIW

5 Alternatives to List Scheduling

5.1 Tie-breaking forward and backward list scheduling

A traditional list scheduler returns a single solution by breaking any ties in the priority of two or more operations arbitrarily. (For example, when our list scheduler finds more than one operation from the ready list with highest priority, it picks the last one in the list.) By running the list scheduler several times and breaking ties randomly, we could potentially generate more and better solutions. Figure 5 is an example from the `tomcatv` benchmark. Assume all load immediates (`LDI`) take one cycle, all add operations (`ADD`) take two cycles, and the copy (`i2i`) takes one cycle. Assume we are scheduling on a machine with two identical functional units. The numbers next to the operations are the priority values that list scheduling uses. In this figure we see two different list schedules that could be generated from the DPG. The second one requires one less cycle. The critical decision comes in the second cycle, where the tie between the `LDId` and `LDIc` must be broken. Scheduling `LDId` early enough results in a shorter schedule.

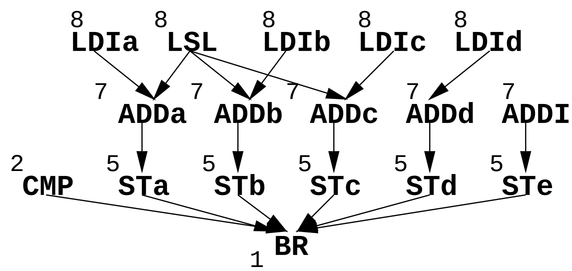
In addition there are some blocks for which a *backward* list scheduler may be able to generate a better solution. A backward list scheduler works by reversing the direction of all edges in DPG, and scheduling the *finish times* of each operation. (Note that the start time of operations must be used to ensure enough available functional units for a given cycle.) This technique tends to cluster operations toward the end of the schedule instead of the beginning like a forward list scheduler. For an example of a block that benefits from backward list scheduling see Figure 6, which shows a block from the `go` benchmark. Assume there are two integer units that can execute the `LDI` operations (one cycle), the `LSL` operation (one cycle), the `ADD` operations (two cycles), `ADDI` operation (one cycle), and the `CMP` operation (one cycle). A separate memory unit executes the `ST` operations (four cycles). All functional units are completely pipelined. A forward list scheduler will schedule the four `LDI` operations and the the `LSL` before scheduling any of the `ADD` operations. This delays the start of the higher latency store operations (`ST`). A better schedule can be found by a backward list scheduler as shown in the example.



Two possible list schedules

```
LDIa  LDIb        LDIa  LDIb
ADDa  LDIc        ADDa  LDId
LDId  i2i         LDIc  ADDd
ADDb  ADDc        ADDb  ADDc
ADDd  LDIe        i2i   LDIe
   ---            JMP
JMP
```

Figure 5 Example block from `tomcatv`



Forward List

LDIa	LSL	
LDIb	LDIc	
LDId	ADDa	
ADDb	ADDc	
ADDd	ADDI	STa
CMP		STb
		STc
		STd
		STe

BR		

Backward List

LDId		
ADDI	LSL	
ADDd	LDIc	
ADDc	LDIb	STe
ADDb	LDIa	STd
ADDa		STc
		STb
		STa

CMP		
BR		

Figure 6 Example block from `go`, showing the benefits of backward list scheduling.

We have implemented a list scheduler that performs 50 forward passes and 50 backward passes breaking ties randomly and keeping the best *static* schedule it finds. We call this technique RBF for Randomized Backward and Forward. The results of this experiment are shown in Tables 2 and 3. The results presented are averages of five different runs (except `fpppp` which was only run once) using different random number seeds. An entry without a decimal point indicates that each of the five runs returned the same result. In about half of the experiments the RBF scheduler found a shorter static schedule for the program which led to a reduced execution time. However, all of the gains were small. It is our belief that one pass list scheduling is finding the shortest schedule most of the time. We intend to develop some experiments to test this hypothesis.

5.2 Iterative Repair

5.2.1 The iterative repair framework

List scheduling has been used for years as the underlying scheduling algorithm for instruction schedulers of different scopes (*e.g.* [24, 10, 15].) Here we introduce the application of a repair based scheduling technique called "iterative repair" to the problem of instruction scheduling in a compiler. We believe this technique shows promise, especially as scheduling problems become increasingly complicated. This algorithm comes from the AI community and is described by Lin and Kernighan [16], and Zweben, *et. al.* [28, 27].

The generalized algorithm is presented in Figure 8. The idea is straightforward. First, create an instruction schedule that begins each operation as early as possible with respect to the precedence constraints of the DPG, but ignores the resource constraints imposed by the limited number of processing elements. Now "repair" the schedule by moving operations that have a resource conflict, to a point later in the schedule. This reduces the number of resource conflicts for the cycle being repaired. A resource conflict is simply a point in the schedule where more operations are scheduled than the available number of functional units.

Benchmark	dynamic		static	
	1 pass List	RBF	1 pass List	RBF
`clean`	4515619	4515223	11429	11421
`compress`	10641037	10638494	1597	1596
`dfa`	696450	696450	1356	1356
`fft`	22469970	22408520	2700	2697
`go`	589209782	589029192	73456	73391
`jpeg`	45900780	45851813	19785	19772.2
`nsieve`	2288889385	2288889380	271	270
`water`	36111497	36110982	6469	6465
`fmin`	5370	5370	712	712
`rkf45`	818884	818884	2387	2387
`seval`	3340	3340	1055	1055
`solve`	2813	2813	1007	1007
`svd`	14649	14649	2481	2481
`urand`	1117	1117	192	192
`zeroin`	4603	4603	545	545
`applu`	884028559	883956398	13334	13315
`doduc`	16953587	16940704	42127	42088
`fpppp`	95701038	95673113	10521	10520
`matrix300`	43073238	43073338	429	429
`tomcatv`	436717483	434116483	953	951

Table 2 RBF Scheduling – Basic Blocks

Benchmark	dynamic		static	
	1 pass List	RBF	1 pass List	RBF
`clean`	3769537	3769144	9973	9967
`compress`	9625986	9625986	1376	1376
`dfa`	594859	594859	1000	1000
`fft`	22150570	22089120	2513	2510
`go`	501223760	500983680.6	54002	53938
`jpeg`	42529297	42479921.6	17480	17466.4
`nsieve`	2254242028	2254242023	249	248
`water`	33651055	33650735	6039	6037
`fmin`	4495	4495	503	503
`rkf45`	761549	761482	2055	2052
`seval`	3283	3283	993	993
`solve`	2671	2671	935	935
`svd`	13963	13961.2	2234	2234
`urand`	1081	1081	172	172
`zeroin`	4095	4095	446	446
`applu`	866954001	866781842	12933	12914
`doduc`	16141047	16130751.4	38536	38502
`fpppp`	90637430	90552395	9798	9794
`matrix300`	42883523	42883523	361	361
`tomcatv`	436707058	434106050	912	910

Table 3 RBF Scheduling – Extended Basic Blocks

The earliest cycle with a conflict is found, and one of the conflicting operations is selected (line (1) in the algorithm). This operation and all operations that depend on it are removed from the schedule (called *unscheduling*). The selected operation and its dependent operations are then inserted back into the schedule (called *rescheduling*) at a later point (line (2) in the algorithm). We continue repairing the schedule until there are no more resource conflicts. The algorithm is run a "user defined" number of times, and the shortest schedule over all the trials is selected as the final schedule.

Iterative repair is a polynomial time algorithm. Several variables affect the running time. Let T be the number of trials, R the number of repairs per trial, and C the complexity of a single repair. Then the the worst case complexity of the algorithm is *O(TRC)*. The parameters R, and C will be discussed for each version of iterative repair that we describe.

It is important to note that not all possible schedules exist in the iterative repair space. Consider the example in figure 7. Let `A`, `B`, `C`, and `D`, be operations with no dependences, and unit latency. Assume a machine with three identical processing elements each capable of executing all the operations. The schedule in part (a) of the figure is the as-early-as-possible schedule used as the iterative repair starting point. Part (b) shows a schedule that iterative repair might generate, and part (c) shows a schedule that can *never* be

	(a)	(b)	(c)
`cycle 1:`	`A B C D`	`A B C`	`A B`
`cycle 2:`		`D`	`C D`

Figure 7 Example showing incompleteness of iterative repair

generated by our formulation of iterative repair scheduling.

Iterative repair provides an important improvement over list scheduling; the tradeoff between performance of the scheduled code and the compilation time is made explicit. Performing more iterations of the algorithm results in a longer compile time, but usually produces a better final schedule. As an example of this see figure 9. In this experiment we used an iterative repair scheduler to schedule a block from `twldrv` in the `fpppp` benchmark. The graph shows the number of iterative repair trials run versus the average length of the schedule found. We ran the scheduler ten times at each trial level with different random number seeds. Another advantage is that the iterative repair algorithm *always* has a complete schedule to look at. Whereas a list scheduler builds up the schedule cycle by cycle, repair-based algorithms start with a complete schedule and repair it.

Different variants of iterative repair scheduling are discussed in the next sections. The framework remains basically unchanged, but different methods of selecting a candidate and rescheduling are explored.

5.2.2 IR scheduling

This variation of iterative repair is designed to run quickly. The candidate (1) is selected randomly, from those involved in the earliest resource conflict. Rescheduling is done by placing the operations back in the schedule at the earliest point where both precedence constraints and resource constraints are satisfied. This method guarantees that the number of conflicts in the schedule will decrease by at least one for each repair. Thus we can perform at most N repairs per trial, where N is the number of operations, and each repair will move at most N operations. Thus the worst case complexity is $O(TN^2)$.

Our preliminary experiments show that this particular method of iterative repair does not perform too well against list scheduling. In fact for all of our benchmark programs, using the IR scheduler increased or

Input: Data Precedence Graph. Parameters of machine (instruction latencies, pipelining, number of functional units, etc.). The number of iterations to perform *iter*.

Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine.

Algorithm:

min = largest integer
shortest is a schedule initially empty
for x = (1 to *iter*)
    Create an initial *schedule* by scheduling all operations as early as possible subject to precedence constraints.
    while (there exist resource conflicts in *schedule*)
        conflict_time = the cycle of the first resource conflict in the schedule
(1)   select an *operation* that has a resource conflict at *conflict_time*
        unschedule *operation* and all its successors in DPG
(2)   reschedule *operation* and its successors later in *schedule*.
    endwhile
    if length of *schedule* is less than *min*
        then *min* = length of *schedule*
        shortest = *schedule*
    endif
endfor

Figure 8 Basic Iterative Repair Scheduling algorithm

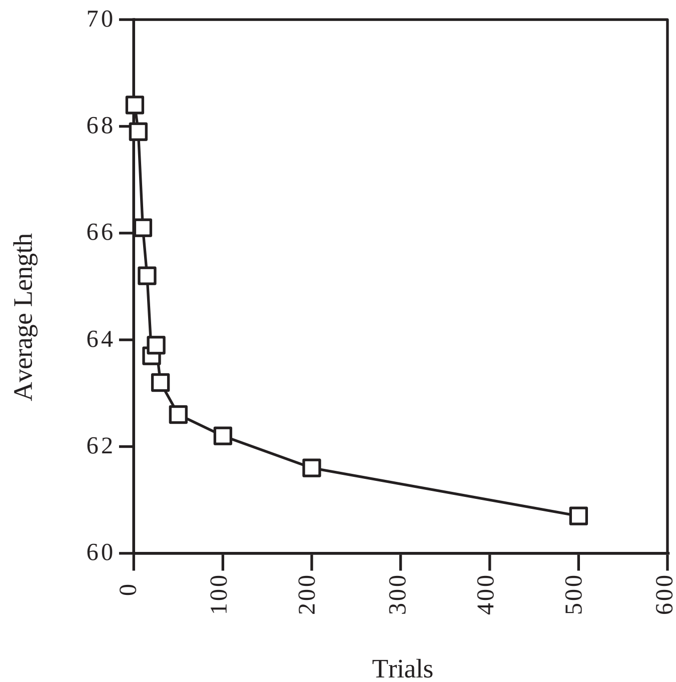


Figure 9 Example showing the time/schedule quality tradeoff of iterative repair.

did not affect the dynamic instruction counts when scheduling over basic blocks or extended basic blocks.

5.2.3 IR-RDF scheduling

The major constraint in IR scheduling is the rescheduling phase. Instead of rescheduling an operation at a point where its resource needs are met, IR-RDF reschedules at the earliest point in the schedule regardless of resource conflicts (while not violating precedence constraints.) In essence each operation that is rescheduled is moved one cycle later in the schedule. A later repair will remove any new conflicts introduced by the rescheduling phase. This technique makes the search space and the solution space much larger, so we must explore it intelligently. In order to facilitate this, we define an *evaluation function* that gets applied to a schedule. The evaluation function maps schedules to real numbers. After each repair, we can evaluate the new schedule and see if it is "better" (using the evaluation function) than the previous one.

Our evaluation function computes the *resource dilation factor* (or RDF) of the schedule after each repair operation [26]. The RDF is computed by looking at each cycle c in the schedule. Let t be a functional unit type. Then, $n(t)$ is the number of functional units of type t available in the given architecture. Let $ops(c,t)$ be the number of operations scheduled to begin at cycle c on functional type t. We then compute $r_{c,t}$, which is the contribution to the RDF of a particular functional unit type t and a particular cycle c, and sum over all cycles and functional unit types. Then the RDF is computed as follows:

$$\sum_c \sum_t r_{c,t} = \begin{cases} n(t) & \text{if } n(t) \leq ops(c,t). \\ ops(c,t) & \text{otherwise.} \end{cases}$$

Alternatively, RDF equals the number of cycles in the schedule multiplied by the number of functional units plus the number of operations that need to be moved due to a resource conflict. The RDF can also be computed as the sum of the operations in the schedule and the number of "holes" (unused functional units) in the schedule. We compute the RDF incrementally, to keep the computation as efficient as possible. After each repair, the new RDF is compared to the previous RDF. If the new RDF is lower, the new schedule is accepted. If the new RDF is higher, the new schedule is rejected stochastically. Currently, we retain a schedule with a higher RDF, ten percent of the time. This is done to avoid local minima. Unfortunately, the worst case running time of this algorithm has increased significantly. Now we may need to perform $O(N^2)$ repairs, for a worse case running time of $O(TN^3)$. We are working on ways to tighten this bound and reduce it.

When scheduling over EBB's, we would like to encourage the scheduler to push operations down in the CFG to potentially less executed areas of the code. To do this we define the *weighted* RBF. Simply put, operations in blocks that are farther down in the control flow graph contribute less to the RDF. Formally, let an EBB by an ordered list of blocks $B_1, \ldots, B_k$ as described in Section 4.2. Then, each $r_{c,t}$ for a given block B_i in the above equation is multiplied by W^i, where $W < 1$ is the *weighting factor*. Ideally, the weighting factor would vary for each block in the EBB, and would be equal to the percentage of times the block executes with respect to the header of the EBB. In our experiments we tried several different fixed weighting factors.

5.2.4 IR-BIAS scheduling

IR-BIAS is built on IR-RDF. We use the same rescheduling technique proposed in the previous section, but we bias the selection of which conflicting operation to move. Before scheduling is performed, each node in the DPG is assigned a priority, with the same heuristic used in list scheduling. Operations with a higher priority are less likely to be selected for moving. Thus, high priority nodes have a better chance of appearing earlier in the schedule.

To perform biased selection we compute the *inverse priority* for each operation at a given cycle. Let *high* be the priority of the highest priority operation for a given cycle. Then the inverse priority of an operation equals *high*, minus the priority of the operation, plus one. These inverse priorities are then summed for the cycle. The probability that an operation gets chosen to be moved is equal to its inverse priority divided by the sum of the inverse priorities for the cycle. For example, assume three operations with priority 1, 5, and 4. The inverse priorities are 5, 1, and 2, respectively. Thus, the probability that the first operation is selected is 5/8, and so on. Biasing does not alter the worst case complexity of the algorithm as described in the previous section, but will increase the constant.

5.2.5 Experimental Results

In this section we give experimental results on the various iterative repair scheduling techniques described above. Table 4 gives dynamic instruction counts using scheduling over basic blocks. Three iterative repair techniques (IR, IR-RDF, and IR-BIAS) are compared to a single forward pass of list scheduling. The iterative repair algorithm was run for 10 or 100 iterations on each basic block, and the shortest *static* schedule was kept. Each test was run five times with different random number seeds, and the results averaged. Any entry that is better than one pass list scheduling is underlined. Notice how more iterations improve the schedules as do the refinements to the basic iterative repair algorithm. In only a few cases did an iterative repair algorithm do better than one pass list (**clean**, **compress**, **fft**, **go**, **nsieve**, **doduc**, and **tomcatv**) and the gains were minimal. Looking at Table 2, iterative repair did no better than RBF. We have found only a single basic block (in the **twldrv** module, from the **fpppp** benchmark) where an iterative repair algorithm found a shorter schedule than RBF.

Static instruction counts are given in table 5. Again, any gains over list scheduling are minimal and in no case did an iterative repair algorithm outperform RBF.

Presumably, the gains are minimal due to the effectiveness of list scheduling over basic blocks. We feel that list scheduling is finding the optimal schedule in most cases, and RBF finding even more. One piece of evidence supporting this is provided by looking at the number of blocks that required repair. That is, in what percentage of the blocks did the as-early-as-possible schedule satisfy the resource constraints of the machine? Both iterative repair and list scheduling can trivially find the shortest schedule for these blocks. Our experiments show that about 77 per cent of all blocks required no repair. This number is a function of the amount of parallelism available in the blocks and on our architecture model. Thus on most of the blocks iterative repair had no chance to find a shorter schedule than list scheduling. We plan to do more experiments to get a better feel for how well list scheduling is doing.

There are several cases (**jpeg**, **applu**, **doduc**, and **fpppp**) where list scheduling did better than any version of iterative repair. Theses benchmarks have several large and very complicated blocks. Even at 100 iterations, the heuristics guiding iterative repair were not adequate to guide the scheduler to the best solution for theses blocks. More iterations (or preferably better heuristics) are needed.

We also did some experiments scheduling over EBB's. The results of these experiments are given in Table 6. In this table we show dynamic instruction counts for one pass list scheduling and various iterative repair schedulers. The column labeled IR-WBIAS, uses IR-BIAS scheduling with the RDF weighting technique described in Section 5.2.3. The weighting factor W was set to 0.75. The results of these experiments are more promising. In twelve of the experiments, some variant of iterative repair found a shorter dynamic schedule than single pass list. Nine of these twelve (**clean**, **compress**, **fft**, **go**, **water**, **rkf45**, **seval**, **svd**, **fpppp**) were also shorter than the RBF scheduled version. (See Table 3).

Table 7 gives static instruction counts for scheduling over EBB's. While an IR algorithm defeated one pass list in several cases, in only two cases (**rkf45**, **fpppp**) did an IR algorithm find a shorter static schedule than RBF. It is also interesting to note that when scheduling over EBB's, the shortest static schedule does not always lead to the shortest dynamic schedule. Let S_1 and S_2 be two different schedules for the same EBB, and let S_1 be shorter than S_2. All other things being equal, S_2 may result in better runtime performance than S_1 if the lengths of the most frequently executed blocks in S_2 are shorter than those in S_1. Some blocks in S_2 must be longer, but if these blocks are not frequently executed the overall affect may be faster code.

To get a feel for the number of solutions that an iterative repair technique can generate see figure 10. A single basic block from **go** was scheduled 100,000 times with an iterative repair scheduler. (The scheduler was similar to IR-RDF, but a repair was accepted regardless of the RDF of the new schedule.) The graph shows the number of solutions of a given length and depth found by iterative repair. (Depth refers to the number

		IR		IR-RDF		IR-BIAS	
Benchmark	List	10	100	10	100	10	100
`clean`	4515619	4528115.4	4520099.4	4516266.0	4515223	4516197.4	4515223
`compress`	10641307	10673958.8	10650581.8	10639061.8	10638494	10638494	10638494
`dfa`	696450	696450	696450	696450	696450	696450	696450
`fft`	22469970	23090574.0	22768971.0	22470729.2	22408521	22408521	22408521
`go`	589209782	590067972.8	589212192.4	589237106.4	589072793.2	589091787.0	589030352.4
`jpeg`	45900780	46712318.2	46522418.0	46313955.6	46180382.6	46124729.0	45945758.6
`nsieve`	2288889385	2288889396.0	2288889386.2	2288889381.0	2288889380	2288889380	2288889380
`water`	36111497	36308195.8	36244544.2	36166711.0	36153042.8	36115836.6	36110982
`fmin`	5370	5379.4	5377	5371.4	5370	5370	5370
`rkf45`	818884	821102.6	818885	818884	818884	818884	818884
`seval`	3340	3375.6	3357.8	3345.4	3340	3340	3340
`solve`	2813	2813	2813	2813	2813	2813	2813
`svd`	14649	14665.2	14656.8	14650.8	14649	14649.4	14649
`urand`	1117	1119.0	1117	1117	1117	1117	1117
`zeroin`	4603	4609	4609	4603.8	4603	4603	4603
`applu`	884028559	904406705.0	896702377.4	891518764.2	888444826.6	885429270.0	885011230.8
`doduc`	16953587	17325084.0	17193255.6	17011929.8	16977174.6	16961180.8	16953588.8
`fpppp`	95701038	108043438.8	105845599.6	102060412.2	96715136.4	96784809.8	96350992.2
`matrix300`	43073238	43073338.0	43073238	43073238	43073238	43073238	43073238
`tomcatv`	436717483	457260285.0	450239624.0	438022076.6	435937183.0	435678103.8	434116483

Table 4 Dynamic instruction counts for various scheduling techniques – Basic Blocks

		IR		IR-RDF		IR-BIAS	
Benchmark	List	10	100	10	100	10	100
`clean`	11429	11447.2	11430.4	11424.6	11421.6	11422	11421
`compress`	1597	1608.4	1602.2	1596.4	1596	1596	1596
`dfa`	1356	1356	1356	1356	1356	1356	1356
`fft`	2700	2718.4	2706.8	2699.4	2698	2698	2698
`go`	73456	73644.6	73464.4	73446.8	73399	73405	73392.0
`jpeg`	19785	19914.4	19855.6	19817.4	19796.2	19794.0	19784.0
`nsieve`	271	282.4	276.2	270.4	270	270	270
`water`	6469	6607.2	6561.6	6492.6	6481.0	6469.6	6465
`fmin`	712	713.4	713	712.2	712	712	712
`rkf45`	2387	2390.4	2388	2387	2387	2387	2387
`seval`	1055	1061.6	1058.2	1056.8	1055	1055	1055
`solve`	1007	1007	1007	1007	1007	1007	1007
`svd`	2481	2484.6	2482.2	2481.4	2481	2481.2	2481
`urand`	192	192.2	192	192	192	192	192
`zeroin`	545	547	547	545	545	545	545
`applu`	13334	13800.8	13625.4	13457.6	13414.4	13354.8	13338.8
`doduc`	42127	42907.2	42624.8	42239.6	42165.4	42130.8	42112.2
`fpppp`	10521	11483.8	11320.2	10954.4	10907.4	10549.8	10533.4
`matrix300`	429	431	429	429	429	429	429
`tomcatv`	953	972.8	966.4	957.4	952.4	953.2	951

Table 5 Static instruction counts for various scheduling techniques – Basic Blocks

Benchmark	List	IR-RDF 10	IR-RDF 100	IR-BIAS 10	IR-BIAS 100	IR-WBIAS 10	IR-WBIAS 100
`clean`	3769537	3794603.6	3786661.8	3777599.4	3776434.0	3771234.6	3765520.6
`compress`	9625986	9629629.0	9628913.0	9627748.2	9627249.0	9626700.2	9625961
`dfa`	594859	595368.2	594859	594859	594859	594859.2	594859
`fft`	22150570	22157730.0	22098849.2	22090657.0	22094753.0	22037461.0	22015393.0
`go`	501223760	502920399.8	502745901.2	502780079.0	501807780.2	499189973.4	498377969.8
`jpeg`	42529297	43054238.2	42887923.4	42856853.2	42652123.6	42827672.6	42657160.6
`nsieve`	2254242028	2254242023.2	2254242023	2254242023	2254242023	2254242023	2254242023
`water`	33651055	34136343.4	34029883.8	33746527.8	33746459.8	33642309.8	33621095.6
`fmin`	4495	4496.6	4495	4495	4495	4495	4495
`rkf45`	761549	761523.4	761658.2	761607.6	761598.0	761379.4	761468.8
`seval`	3283	3290.2	3283.4	3283.6	3284	3280.6	3279.6
`solve`	2671	2671	2671.2	2671.2	2671	2671	2671
`svd`	13963	13963.0	13961.0	13960.6	13961.0	13958.8	13958
`urand`	1081	1117.0	1105.0	1099.0	1099.0	1081	1081
`zeroin`	4095	4095.4	4095	4095	4095	4095	4095
`applu`	866954001	873379085.4	871649889.6	868565137	868124958.4	868642169.2	868008358.6
`doduc`	16141047	16244608.0	16193940.6	16163013.0	16143007.4	16146709.6	16136166.8
`fpppp`	90637430	93419117.6	92685310.2	90083006.2	88748044.4	88190167.0†	87808224.4†
`matrix300`	42883523	42883523	42883523	42883523	42883523	42883603.0	42883523
`tomcatv`	436707058	439050002.0	436188890.0	434367170.0	434106054.0	436191946.0	434366138.0

† in this experiment the `fpppp` module was compiled with list scheduling, and the other modules as indicated.

Table 6 Dynamic instruction counts for various scheduling techniques – Extended Basic Blocks.

Benchmark	List	IR-RDF 10	IR-RDF 100	IR-BIAS 10	IR-BIAS 100	IR-WBIAS 10	IR-WBIAS 100
`clean`	9973	9977.8	9970.4	9970.0	9967.2	9974.6	9969.4
`compress`	1376	1377.8	1376.2	1376.4	1376	1377.4	1376
`dfa`	1000	1000.4	1000	1000	1000	1000.2	1000
`fft`	2513	2513.6	2511.2	2511.2	2511	2513.4	2511
`go`	54002	54128.2	53995.2	53993.0	53954.0	54149.6	54057.8
`jpeg`	17480	17517.8	17493.4	17492.8	17479.6	17506.4	17489.4
`nsieve`	249	248.2	248	248	248	248	248
`water`	6039	6076.6	6056.0	6044.0	6039.6	6054.6	6046.2
`fmin`	503	503.4	503	503	503	503.2	503
`rkf45`	2055	2051.6	2051	2051	2050	2052.4	2051.6
`seval`	993	994.0	993	993	993	994	994
`solve`	935	935	935	935	935	935	935
`svd`	2234	2234.2	2234	2234	2234	2234	2234
`urand`	172	172	172	172	172	172	172
`zeroin`	446	446.2	446	446	446	446	446
`applu`	12933	13069.4	13023.6	12960.6	12941.6	12968.4	12946.2
`doduc`	38536	38685.6	38600.8	38551.0	38528.6	38578.0	38546.0
`fpppp`	9798	10131.6	10088.0	9784.4	9742.2	9744.4	9737.2
`matrix300`	361	361	361	361	361	361.2	361
`tomcatv`	912	915.2	912.0	910.4	910	913.0	910.2

Table 7 Static instruction counts for various scheduling techniques – Extended Basic Blocks

of repairs needed to find the solution.) Over 50,000 unique solutions were generated by the iterative repair algorithm. In contrast, our list scheduler, augmented with random-tie breaking, found only 72 solutions.

5.2.6 Future Work

We believe that iterative repair is a promising scheduling technique. We feel that as scheduling problems become more complicated, any list scheduling heuristic becomes less and less effective. Thus, more opportunity may be available to randomized approaches like iterative repair. We see a glimpse of this phenomenon when moving from scheduling basic blocks to scheduling extended basic blocks.

There are several things to be explored concerning iterative repair schedulers. First, we will investigate how well list scheduling is actually performing on our set of benchmarks. This will give us an estimate of how much opportunity is available for an iterative repair scheduler to improve the program. Second, we can make changes to the iterative repair algorithm itself. We have detailed the refinements made to the algorithm so far, and plan to investigate others. One possibility is the use of simulated annealing. Currently after repairing a schedule, we accept a new schedule with a higher RDF ten percent of the time. The simulated annealing strategy dictates that the probability of a "worse" schedule being accepted decreases over time. Thus at the beginning of the repair process, the selected repair is chosen more randomly than at the end. This technique has shown promise with other scheduling problems. Finally we will examine the conditions under which iterative repair performs well and those under which it performs poorly. We will investigate how various processor features affect the performance, and attempt to understand what features of a particular instance of an instruction scheduling problem affect the different schedulers.

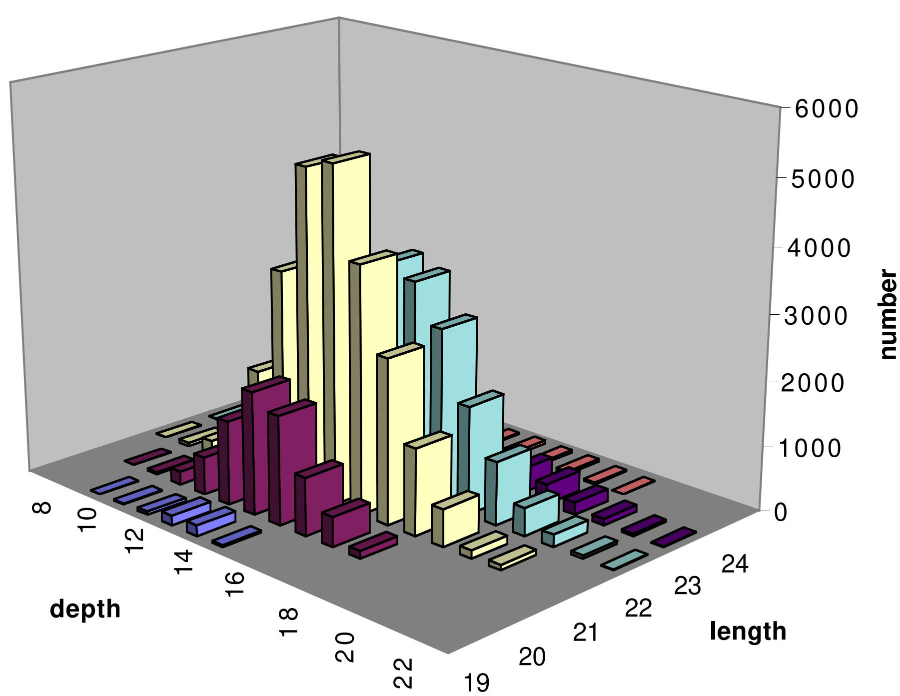


Figure 10 Solutions found by iterative repair on a block from `go`

6 Interaction Between Instruction Scheduling and Register Allocation

There has been a lot of work done recently on the interplay between register allocation and instruction scheduling. These two compiler phases are often antagonistic; an improved schedule may make for more spill code, but a better allocation may unnecessarily inhibit instruction level parallelism. Thus the question arises: Is it better to do register allocation first or instruction scheduling first, or to combine the two in some way?

Figures 12 through 14 show an example taken from the CRAIG paper that illustrates the allocation-scheduling tension [4]. Assume a machine with two identical functional units, where loads (LD) and stores (ST) take two cycles, and all other operations take one cycle. The machine has four general purpose registers. Figure 12 shows intermediate code that could be generated by the source code in figure 11.

```
x_pos = x_pos + (x_vel * t);
y_pos = y_pos + (y_vel * t);
```

Figure 11 Source Code

```
 1: LD [x_pos] => r0
 2: LD [x_vel] => r1
 3: LD [t]     => r2
 4: MULT r1 r2 => r3
 5: ADD r0 r3  => r4
 6: ST [x_pos] r4
 7: LD [y_pos] => r5
 8: LD [y_vel] => r6
 9: MULT r6 r2 => r7
10: ADD r5 r7  => r8
11: ST [y_pos] r8
```

Figure 12 Original Intermediate Code

We see that operations 7 through 11 are basically independent of operations 1 through 6, thus these two sets of operations can be run in parallel. This is reflected in Figure 13 which shows the result of first scheduling the code, and then performing register assignment. The resultant code executes in seven cycles and requires five registers (`r0` - `r4`). Since the machine has only four registers, spill code would need to be inserted.

```
LD [x_vel] => r1  |  LD [t]     => r2
LD [x_pos] => r0  |  LD [y_vel] => r6
MULT r1 r2 => r3  |  LD [y_pos] => r5
ADD r0 r3  => r4  |  MULT r6 r2 => r7
ST [x_pos] r4     |  ADD r5 r7  => r8
ST [y_pos] r8
NOP
```

(a) After scheduling

```
LD [x_vel] => r0  |  LD [t]     => r1
LD [x_pos] => r2  |  LD [y_vel] => r3
MULT r0 r1 => r0  |  LD [y_pos] => r4
ADD r0 r2  => r0  |  MULT r3 r1 => r1
ST [x_pos] r0     |  ADD r4 r1  => r1
ST [y_pos] r1
NOP
```

(b) After allocation

Figure 13 Code scheduled then allocated

Figure 14 shows the result of performing allocation first and then scheduling. Notice the many new dependences created by the early allocation. Some important ones to note are the anti-dependences from operation 8 to 5, and from 7 to 6. This code takes nine cycles to execute but uses only three registers (`r0` - `r2`.) Note that different allocations can ultimately result in different schedules. (To see this, look at the presentation of the example in the paper [4].)

```
 1: LD [x_pos] => r0
 2: LD [x_vel] => r1
 3: LD [t]     => r2
 4: MULT r1 r2 => r1
 5: ADD r0 r1  => r0
 6: ST [x_pos] r0
 7: LD [y_pos] => r0
 8: LD [y_vel] => r1
 9: MULT r1 r2 => r1
10: ADD r0 r1  => r0
11: ST [y_pos] r0
```

(a) After allocation

```
LD [x_vel] => r1  |  LD [t]     => r2
LD [x_pos] => r0  |  NOP
MULT r1 r2 => r1  |  NOP
ADD r0 r1  => r0  |  LD [y_vel] => r1
ST [x_pos] r0     |  LD [y_pos] => r0
MULT r1 r2 => r1  |  NOP
ADD r0 r1  => r1  |  NOP
ST [y_pos] r1     |  NOP
NOP               |  NOP
```

(b) After scheduling

Figure 14 Code allocated then scheduled

A technique that attempts to solve the two problems at the same time, may find a solution like that in figure 15. This schedule uses four registers and executes in eight cycles.

```
LD [x_vel] => r0  |  LD [t]     => r1
LD [x_pos] => r2  |  LD [y_vel] => r3
MULT r0 r1 => r0  |  NOP
ADD r0 r2  => r0  |  LD [y_pos] => r2
ST [x_pos] r0     |  MULT r3 r1 => r1
ADD r2 r1  => r1  |  NOP
ST [y_pos] r1     |  NOP
NOP               |  NOP
```

Figure 15 A combined solution

Typically the instruction scheduling and register allocation phases are completely distinct; neither phase pays any concern to the requirements or restrictions of the other. Thus, we have the phase ordering problem illustrated in the above example. Determining which phase to run first is largely dependent on the target architecture and the code being compiled. In the above example, if the target machine has sufficient registers we would be happy with the code in Figure 13. However if that code causes too much spill code to be generated, the code in Figure 14 may be preferable.

There are several different approaches for looking at this problem in the literature. Several authors have looked at the phase ordering problem [3, 6, 11, 22].

There have been several methods developed for integrating the register allocation and instruction scheduling phases. The amount of integration varies from making one phase sensitive to the needs of the other to completely combining the two phases. We will not try to categorize the level of integration here, but rather give a brief overview of the different ideas.

Goodman and Hsu use a scheduler that is sensitive to register pressure [11]. If the number of available registers falls below a particular threshold while scheduling, a different scheduling algorithm is invoked. This algorithm schedules with the goal of minimizing register usage. Once the number of available registers is sufficiently high, the original scheduling algorithm is used. Another method they use, modifies the "width" of the DPG used by the scheduler, during register allocation. By using the DPG, the register allocator has better information about the dependences it will insert. After allocation, an "unmodified" scheduler is used.

In the work of Bradlee, *et.al.*, the method of Goodman and Hsu is compared to a complicated three pass system called RASE [3]. The first pass of RASE is a prescheduler that computes the length of the schedule for each basic block, for a range of register limits. This information is used by global register allocator (the second phase), when deciding what register limit to impose on each basic block. The final phase schedules each block, and makes *local* register assignments.

Norris and Pollock modify a Briggs style allocator [5] to use information from the DPG to limit the number of dependences it introduces [18] . They add edges to the interference graph to reflect all possible code reorderings, that satisfy the DPG for a basic block. They discuss several heuristics for adding these edges, and look at when to add them during the allocation process. Later on some of the interference graph edges may be removed to aid in coloring and reducing spilling, at the cost of adding additional edges to the DPG.

Pinter's work combines information from a modified form of DPG, and the register interference graph, to create the *parallel interference graph* [19]. First the transitive closure of the DPG is taken, and all edges become undirected. Then, edges are added between operations that have a resource constraint imposed by the architecture. Edges in the complement of this graph, represent operations that could be run in parallel. This new interference graph takes into account all the possible parallelism in the program.

Brasier, *et. al.* have done work on CRAIG (Combining Register Assignment and Instruction Scheduling) [4]. First, CRAIG attempts to perform scheduling before register allocation (**late** register assignment). If spill code is required, CRAIG performs a round of register allocation followed by scheduling (**early** register assignment). If no spilling was required, CRAIG combines the two interference graphs by adding edges from the **late** interference graph to the **early** interference graph. Adding the appropriate edges makes that graph harder to color, but allows more parallelism to be exploited. This new graph is used for register assignment, and the resultant code is then scheduled.

Motwani, *et.al.* provide NP-completeness results for a simple instance of CRISP (Combined Register Allocation and Instruction Scheduling Problem) [17]. They also describe an algorithm for combining the problems when scheduling/allocating for a basic block, called the (α,β)-Combined Heuristic. They define a value called the *register rank* for each operation. This value is analogous to the list scheduling priority described above, except that it is used to minimize register pressure instead of schedule length. The register rank and a traditional list scheduling priority are summed together (weighted by the coefficients α and β), to create a new list scheduling priority value. The best α and β are determined experimentally. List scheduling is then performed in the usual way, using the new priority values. Register allocation is then performed on the block.

We are investigating ways to combine the two problems within our iterative repair scheduling framework. One way is to treat registers as an additional resource that needs to be managed during the scheduling process. Another alternative is to use some estimate of register pressure in the evaluation of schedules produced by iterative repair. Preliminary experiments indicate that iterative repair schedulers find a large number of schedules and that some of these schedules may have better register pressure properties.

The graph in Figure 16, characterizes the 50,000+ unique solutions found by 100,000 iterations of an iterative repair scheduler on a single basic block from the `go` benchmark. Schedule length is shown on the x-axis and the number of required registers on the y-axis. Notice that the best solutions have length 19 and require seven registers. All the list schedules we were able to generate also had length 19 but required eight registers. In addition, there are many solutions of length 20 that require only five registers. List scheduling discovered none of them.

We also did an experiment on about 3700 basic blocks from our benchmarks. We ran two versions of the scheduler on each block and noted the schedule that had the lowest register pressure. The first scheduler

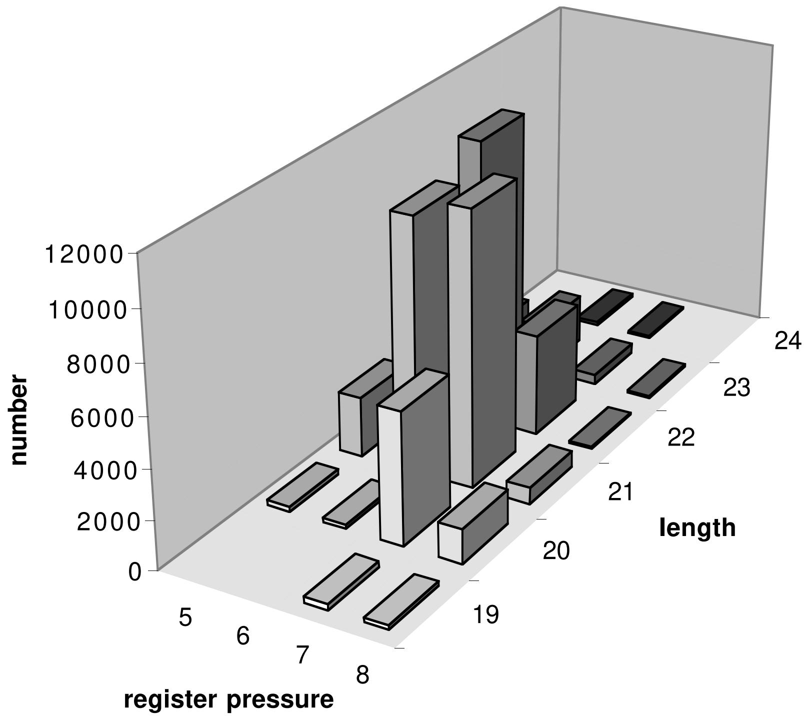


Figure 16 Register pressure for different schedules in block from `go`

was a list scheduler that we ran 2500 times, breaking ties randomly. The second scheduler was IR-RDF as described above, run 5000 times. (The difference in trials is due to the much larger solution space searched by the iterative repair scheduler.) In roughly 14 per cent of the basic blocks, the iterative repair scheduler found some schedule (ignoring schedule length) with lower register pressure than any found by the list scheduler.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Notices*, 26(6):241–255, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[3] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Systems*, pages 122–131, April 1991.

[4] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. CRAIG: A practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[6] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

[7] Jr. E. G. Coffman, editor. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[8] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, 1986.

[9] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[10] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[11] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, July 1988.

[12] Brett L. Huber. Path-selection heuristics for dominator-path scheduling. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, Michigan, 1995.

[13] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscaler compilation. *Journal of Supercomputing – Special Issue*, 7:229–248, July 1993.

[14] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[15] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, pages 261–294, September 1980.

[16] S. Lin and B. Kernighan. An effective heuristic for the traveling salesman problem. *Operations Research*, 21, 1973.

[17] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Report TR 698, Courant Institute, July 1995.

[18] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. *Proceedings of Supercompting '93*, pages 804–813, November 1993.

[19] Schlomit S. Pinter. Register allocation with instruction scheduling: A new approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[20] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing – Special Issue*, 7:9–50, July 1993.

[21] Philip H. Sweany. *Inter-Block Code Motion without Copies.* PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.

[22] Philip H. Sweany and Steven J. Beaty. Post-compaction register assignment in a retargetable compiler. *SIGMICRO Newsletter*, 21(12):107–116, December 1990. *Proceedings of the 23rd Annual Workshop in Microprogramming and Microarchitecture (MICRO-23).*

[23] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling – A global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, December 1992. *Proceedings of the 25th Annual International Symposium on Microarchitecture.*

[24] Roy F. Touzeau. A Fortran compiler for the FPS-164 Scientific Computer. *SIGPLAN Notices*, 19(6):48–57, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[25] Jeffrey D. Ullman. Complexity of sequencing problems. In E. G. Coffman, editor, *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[26] W. Zhang and T. G. Dietterich. A reinforcement learning apporach to job-shop scheduling. In *Proceedings of IJCAI-95*, 1995.

[27] Monte Zweben, Eugene Davis, Brian Daun, and Michael J. Deale. Scheduling and rescheduling with iterative repair. IEEE Transactions on Systems, Man, and Cybernetics, 23(6):1588–1596, 1993.

[28] Monte Zweben, Eugene Davis, Brian Daun, Ellen Drascher, Michael Deale, and Megan Eskey. Learning to improve constraint-based scheduling. *Artificial Intelligence*, 58:271–296, 1992.