Backtracking-based Instruction Scheduling To Fill Branch Delay Slots

Ivan D. Baev and Waleed M. Meleis
Electrical and Computer Engineering Department
Northeastern University, Boston, MA
meleis@ece.neu.edu baev@ece.neu.edu

Santosh G. Abraham
Sun Microsystems
901 San Antonio Road, Palo Alto, CA
santosh.abraham@eng.sun.com

Abstract

Conventional schedulers schedule operations in dependence order and never revisit or undo a scheduling decision on any operation. In contrast, backtracking schedulers may unschedule operations and can often generate better schedules. This paper develops and evaluates the backtracking approach to fill branch delay slots. We first present the structure of a generic backtracking scheduling algorithm and prove that it terminates. We then describe two more aggressive backtracking schedulers and evaluate their effectiveness.

The full-backtracking OperBT scheduler enables backtracking for all operations and unschedules already scheduled operations to make space for the current operation. For the SPECint95 benchmark, the OperBT scheduler increases the percentage of superblocks scheduled optimally over a conventional non-backtracking scheduler from a geometric mean of 66.9% to 81.4%, an increase of 21.7%. The selective backtracking ListBT scheduler enables backtracking only when scheduling certain types of operations for which backtracking is likely to be advantageous. This hybrid scheduler is almost as good as the OperBT scheduler in terms of generated schedule length but backtracks about four times less often. We conclude that aggressive backtracking-based instruction schedulers can effectively improve schedule quality by eliminating branch delay slots with a small amount of additional computation.

Keywords: instruction scheduling, global scheduling, compiler optimization, EPIC, VLIW, instruction-level parallel processors

---

A preliminary version of this paper appeared in the Proc. of the 2000 Intl. Conf. on Parallel Architectures and Compiler Techniques.

1 Introduction

With the increasing demand for high performance processors for media-intensive applications and the improvements in the underlying semiconductor technology, processors support increasing levels of instruction-level parallelism (ILP). Compilers that detect, exploit, and match the available levels of parallelism in these applications to the parallelism supported by the processor are an essential component of the overall solution. In particular, scheduling technology plays a key role in the effective compilation of applications to ILP processors. Instruction scheduling reorders the operations in a scheduling region and packs them into instructions that match the available ILP in the processor. Superscalar processors have hardware to dynamically pack instructions that can be issued in each cycle. In contrast, EPIC (Explicitly Parallel Instruction Computing) [1] and VLIW (Very Long Instruction Word) processors rely on the compiler to statically pack operations to be issued in each cycle. In this paper we discuss instruction scheduling technology for EPIC/VLIW processors, even though we believe that similar techniques can be profitably used in superscalar processors as well.

Following EPIC terminology, *operations* correspond to RISC-style instructions, and *instructions* are a group of operations that issue in a particular cycle. A scheduler schedules individual operations to issue in certain cycles and use certain resources. A conventional scheduler does not undo or revisit scheduling decisions that were made previously. This paper develops and evaluates *backtracking schedulers*, schedulers that can undo previous scheduling decisions and reschedule operations. We believe major trends in processor design highlight the need for such backtracking schedulers.

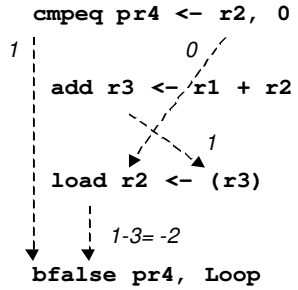
The first trend is toward deeper pipelines. Even a simple RISC processor may have five pipe stages for instruction fetch, align, decode/register read, execute, and write back, respectively. Modern superscalar or VLIW processors may have as many as 15 or 20 stages. In these processors a branch is resolved in the register read or execute phases. The true latency of the branch is the length of the pipeline to the point where the branch is resolved. Processor designs attempt to hide the latency of the branch by predicting the branch. But if the branch prediction is wrong, the early stages of the pipeline have to be emptied causing a misprediction penalty of as much as 10 cycles. A

supplement and/or alternative to branch prediction is to expose all or some of the latency of the branch to the compiler and enable the compiler to fill the delay slots of the branch.

The second trend is toward power-sensitive processor designs for mobile applications. Effective branch prediction hardware requires large memories/caches to maintain a sufficient amount of branch behavior history. The power consumption of large, frequently accessed memories is high and can account for a significant fraction of total on-chip power consumption. Therefore, exposing the branch latency and reducing/eliminating prediction hardware presents an attractive alternative for mobile processor designs.

Branch latency, like other scheduling constraints, is exposed to the scheduler through a dependence graph [2] in which nodes represent operations. An edge from operation *x* to *y* with weight *w* indicates that operation y must be scheduled no earlier than *w* cycles after *x*. If *w* is negative, then *y* must be scheduled no earlier than *–w* cycles before *x*. Consider the control dependence edge from an operation *x* to a succeeding branch *b* in such a graph. The latency of edge *(x, b)* must guarantee that the branch does not transfer control to another scheduling region before *x* completes its work. For example, if the latencies of *x* and *b* are 1 and 3, respectively, then the latency of *(x, b)* is negative, -2, permitting *x* to descend below *b* into the two branch delay slots. The negative latency properly forces the branch to be issued no more than 2 cycles before the issue time of operation *x*.

Unfortunately, conventional list schedulers [3, 4] cannot deal with the explicit branch latency and fill branch delay slots. These schedulers change no scheduling decisions; the highest priority operation is first scheduled, the next highest priority operation is then scheduled, and so on until the last, lowest priority operation is scheduled. To guarantee a legal schedule, conventional schedulers must process operations in a dependence order, traversing the dependence graph forward or backward. Priority is used to choose a particular dependence order. In the dependence graph of a basic block, the branch is a successor of all operations, and therefore a forward-based scheduler considers it for scheduling after all other operations have been scheduled. At that moment, however, we may not be able to schedule the branch so that its delay slots are filled. We cannot simply assign a higher priority to the branch and process it before the lower priority operations with the hope that these operations will fill the branch delay slots. Without backtracking if there are too many lower priority operations we may end up scheduling an operation after the branch delay slots, that is, producing an illegal schedule. For basic block scheduling, this problem can be solved



Conventional scheduler

```
0: cmpeq pr4 <- r2, 0
1: add r3 <- r1 + r2
2: load r2 <- (r3)
3: bfalse pr4, Loop
4: <empty delay slot>
5: <empty delay slot>
```

Backtracking scheduler

```
0: cmpeq pr4 <- r2, 0
1: bfalse pr4, Loop
2: add r3 <- r1 + r2
3: load r2 <- (r3)
```

Figure 1: Backtracking schedulers can fill branch delay slots

by backward (bottom-up) scheduling [5]. Such a scheduler places the only branch in the schedule first, and it then proceeds to place the other operations as late as possible. For superblock and larger scheduling regions with many branches, this approach still fills the delay slots associated with the last branch, however it does not successfully fill delay slots of earlier branches.

Consider the scheduling example in Figure 1 where we assume a single-issue processor with an arithmetic operation latency of one, `load` latency of one, and a branch latency of three. On the left hand side, we show the operations in the body of a while loop that scans to the end of a linked list. The `cmpeq` operation sets the predicate register `pr4` to `true` if register `r2` is 0. The `bfalse` operation branches to the `Loop` label if the register `pr4` is `false`. The edges between operations are labeled with their associated latencies. Certain redundant transitive edges are not shown. The edge from the `load` to the `bfalse` indicates that the live-out value of `r2` must be generated before control is transferred to another block. This edge latency is the difference between the `load` latency of one and the `bfalse` latency of three. The `add` and `load` operations can descend below the `bfalse` into the branch's delay slots. But conventional schedulers schedule instructions cycle by cycle and make sure that all predecessors of an operation are scheduled before the operation. As a result, the `bfalse` is scheduled after all other operations, and consequently its delay slots are unfilled as shown in the middle column. The example also demonstrates a backtracking scheduler. Here the `bfalse` has a higher priority than the `add` and displaces the scheduled `add` which in turn displaces the scheduled `load` from cycle 2 to cycle 3. Both the delay slots of the `bfalse` are now filled. The schedule length is reduced from six to four, for a reduction in cycles of 33%. Note also that merely increasing the priority of the branch operation will not allow an optimal schedule to be produced since conventional schedulers process operations in dependence order.

The potential improvement due to backtracking can be determined by using a tight lower bound on the schedule length. Such bounds were introduced by Rim and Jain [6] and Langevin and Cerny [7] and applied to superblock scheduling in Eichenberger and Meleis [8]. The gap between the bound and the number of dynamic cycles in a schedule produced by a conventional scheduler represents the size of potential improvement. However, a portion of the gap is due to the difference between the lower bound and the number of cycles required by an optimal schedule, and this part of the gap is therefore unrealizable. For the SPECint95 benchmarks scheduled on a set of EPIC processors with the branch latency of one, two, or three cycles, we found that the maximum gap varies between 5.6% and 9.7%.

The trend toward wide-issue processors further accelerates the need for backtracking schedulers. Media-intensive applications have large amounts of parallelism that can be effectively exploited by processors that issue many operations in each cycle. However, wide-issue processors also require a commensurate increase in the number of register read and write ports. Increasing the number of write ports is especially difficult and expensive. The clustered approach partitions register files so that each cluster of functional units accesses its own set of register files. This approach requires compiler techniques to map operations to clusters and may not be effective for some applications because of excessive inter-cluster communication. An alternative is to allow functional units to share write ports. This leaves the compiler responsible for scheduling operations so that there is no resource conflict on the write ports, even among operations with disparate latencies.

Though backtracking schedulers can be more effective than conventional schedulers for a variety of reasons, this paper focuses on how backtracking schedulers can fill branch delay slots. An alternative to backtracking in this respect is peephole optimization. For example, a post-scheduling peephole strategy for filling delay slots goes through the schedule and first identifies branches whose delay slots are unfilled. It then attempts to swap such branches with other regular operations scheduled in earlier cycles without violating dependence and resource constraints. Such peephole optimizations tend to work well only when developed for a particular design with a fixed set of latencies and resources. In a world with a multitude of customized designs for different applications, it may not be cost-effective to develop such specialized compiler optimizations. Even though more systematic techniques exist that support the ability to be retargeted, the local scope of peephole optimizations remains a problem. As the

size of the scheduling regions and the range of scheduling motions increase, peephole optimization becomes less effective [9].

In summary, deeper pipelines and the power requirements of prediction hardware motivate exposing all or some of the branch latency to the compiler. Current non-backtracking instruction schedulers do not generate schedules that consistently fill delay slots and handle resource conflicts effectively. Peephole optimizations are relatively expensive to implement for each processor design under consideration and are not effective in performing larger scale code motions that are required. Backtracking schedulers are an attractive alternative and have proven useful in modulo scheduling [10]. However, backtracking schedulers for acyclic regions have not been developed and their efficacy and runtime complexity have not been previously studied.

In this paper, we develop and evaluate the backtracking approach for acyclic instruction scheduling and show that aggressive backtracking-based schedulers can improve schedule quality by eliminating branch delay slots with a small amount of additional computation. The backtracking technique can be applied on top of any conventional scheduler. Namely, a conventional scheduler defines some operation priorities, attempts to schedule operations in that priority order, and then backtracks when necessary. In this way, we believe that backtracking can be used to improve any conventional scheduling algorithm. Finally, profitable combinations of acyclic and cyclic scheduling algorithms exist but their study is not subject of this work.

The organization of the paper follows. The next section describes the overall scheduling model consisting of a processor description, scheduler input and output. In Section 3, we introduce a generic backtracking scheduling algorithm and prove that it terminates. We then describe two more aggressive backtracking schedulers. The OperBT scheduler is a full backtracking scheduler that attempts to schedule operations in priority order. This scheduler successfully fills branch delay slots but unschedules operations repeatedly and has relatively high runtimes. The ListBT scheduler is a selective backtracking scheduler that backtracks only when it is likely to be profitable. This scheduler is almost as effective in filling branch delay slots but has better runtimes than the full backtracking scheduler. We present a detailed evaluation of both schedulers in Section 4. The related work and conclusions are given in Sections 5 and 6, respectively.

Table 1: Processor configurations: functional units and branch latencies

Processor configuration	Integer units	Floating-point units	Load/store units	Branch latency	Maximum issue width
111L1	1	1	1	1	3
211L1	2	1	1	1	4
111L2	1	1	1	2	3
211L2	2	1	1	2	4
111L3	1	1	1	3	3
211L3	2	1	1	3	4

2 Scheduling Model

2.1 Processor architecture

We use a family of VLIW processors based on the HPL-PD architecture [11]. Each processor has a set of integer, floating-point and memory (load/store) units. A particular processor is described concisely as, say, a 312 processor, indicating that it can issue up to three integer operations, one floating-point operation, and two memory operations in a cycle. Each instruction consists of a set of operations, where each machine operation is a RISC-style operation with source and destination operands. Each instruction may contain several operations of a certain type up to the number of units of that type. We assume that functional units are fully-pipelined, so operations from different instructions (necessarily issued in distinct cycles) do not compete for resources.

In addition, a processor can issue a branch operation in each cycle on one of the integer units. The branch latency is varied from 1 through 3 and the concise notation for a particular processor design encodes the branch latency as a suffix, e.g. 312L2 denotes a 312 processor with a branch latency of two. The latencies of all other operations are fixed as follows: integer ALU 1, float add 3, int/float multiply 3, int/float divide 8, load 2, and store 1. Table 1 describes the variable parameters of six processors that we will use throughout this paper.

2.2 Scheduler input

We first use the IMPACT compiler from the University of Illinois, a part of the Trimaran compiler infrastructure [, 1998 #3], to generate an intermediate representation of the application that is in aggressively optimized superblock form. A *superblock* is a linear chain of basic blocks with a single entry and exits at each branch operation of the individual basic blocks [12]. The IMPACT compiler performs traditional global optimizations, unrolls loops up to eight times, forms superblocks, and applies ILP optimizations to each superblock. The memory disambiguation information computed by the compiler is part of the input. In addition, the input code contains profile information:

each superblock is annotated with weights indicating how often each superblock is executed and how often each branch is taken when the benchmark is run on its data set.

The Elcor compiler from HP Laboratories, also a part of Trimaran, takes the input in superblock form, performs data-flow analyses, and constructs dependence graphs. An edge between two operations is annotated with a latency indicating the minimum separation in their issue times. Data-flow, -anti, and -output dependence edges arise from constraints between the production/consumption of values between operations. In addition, branch operations are associated with control dependences.

2.3 Scheduler output

The scheduler assigns a valid issue cycle for each operation in the superblock. The generated schedule must satisfy the following constraints:

1. dependence edge constraints are satisfied, i.e., for each dependence edge the difference between the issue cycles of the destination and source operations is not less than the edge latency. In addition, the scheduler orders the operations within each cycle so that all zero-cycle dependencies flow from left to right.

2. resource constraints are satisfied, i.e., in our simplified processors, the number of operations of a certain type scheduled in a particular cycle does not exceed the number of units of that type. Also, at most a single branch is scheduled in a cycle and the number of integer operations plus branches does not exceed the number of integer units.

The scheduler optimizes the profile-weighted execution time of each superblock. The superblock execution time is obtained by summing up the contributions of each of its branches where the contribution of a branch *b* is the product of the profile weight of *b* and its exit time. The profile weight of *b* is the number of times the branch is taken during profiling, and the exit time of *b* is the sum of *b*'s issue cycle and the branch latency.

The performance measurements that we report are based on profiling information and not based on actually simulating the scheduled code. This performance measure does not account for stalls caused by cache misses, branch mispredictions, TLB misses, etc. These other factors are expected to be similar with or without backtracking schedulers. Further, we do not use a separate training run using a training data set to generate profile information and another run on an evaluation data set to evaluate the schedules generated. All schedulers evaluated in this paper

use profiling information in an identical manner to generate static priorities for operations. Therefore, we expect that using a single run for both training and evaluation affects the performance of all schedulers in a similar manner.

3 Backtracking schedulers

In this section, we describe a basic backtracking scheduling algorithm and prove that it terminates. We then present two more aggressive backtracking schedulers. The full-backtracking OperBT scheduler enables backtracking for all operations and unschedules already scheduled operations to make space for the current operation. In contrast, the selective backtracking scheduler ListBT enables backtracking only for certain types of operations where backtracking is likely to be advantageous. We begin by briefly discussing the structure of conventional, non-backtracking, schedulers.

A *ready operation* is an operation all of whose predecessors (immediate and non-immediate) have been scheduled. Conventional schedulers process operations in dependence order, i.e., at any scheduling step they consider only ready operations. A *cycle-based* scheduler fills the current cycle with operations before moving on to the next cycle. At any cycle, it schedules the highest priority ready operation that has no resource conflicts with previously scheduled operations. An *operation-based* scheduler considers ready operations in priority order. It assigns the highest priority ready operation the earliest cycle such that its resource and dependence constraints are satisfied. We evaluate the proposed backtracking schedulers by comparing them to a cycle-based one, the Cycle scheduler. Details about its implementation have been previously described [9].

Scheduling by priority in can be beneficial since some operations are more important then others and, intuitively, the important ones should be considered earlier in the scheduling process. However, cycle- and operation-based schedulers do not attempt to schedule a high priority operation if it is not ready. Backtracking schedulers lift this restriction: a high priority operation can be processed even if some of its predecessors have not been scheduled. The highest priority operation is scheduled such that both resource constraints and dependence constraints with respect to previously scheduled operations are satisfied. When no cycles are available to schedule a given operation, one or more operations are unscheduled to make room for the new operation. Such backtracking schedulers are the focus of this paper.

3.1 A basic backtracking algorithm

In this subsection, we describe the structure of a basic backtracking algorithm, BasicBT, and prove that it terminates. Our backtracking schedulers and the Cycle scheduler begin by computing static early and late times as well as a priority for each operation. The *static early time* of an operation is the earliest cycle that it can be issued on a processor with infinite resources. The *static late time* of an operation with respect to a branch *b*, *late*(*b*), is the latest cycle that the operation can be issued on a processor with infinite resources while still issuing *b* at its static early time. Let *maxheight* be the maximum static early time among all operations in the superblock, and the *height* of an operation with respect to a branch *b* equals *maxheight* – *late*(*b*). Intuitively, *maxheight* and *height* represent the length of the longest path in the superblock dependence graph and the length of the longest path from the operation to the branch, respectively.

We set the priority of an operation equal to the sum over all superblock branches, *b*, of the product of the profiled weight of *b* and the height of the operation with respect to *b* [13]. Although the Elcor compiler supports several priority functions, all the evaluations reported here are based on this weighted height priority function. For superblocks, the weighted height priority is a natural extension of the critical path priority, which has been successfully used in basic block scheduling.

For backtracking schedulers, in addition, the *(dynamic) early time* of an operation is the maximum of its static early time and the earliest cycle the operation can be issued on an infinite resource processor while satisfying all dependence edges from previously scheduled predecessors of the operation. The *(dynamic) late time* of an operation is the latest time the operation can be issued on an infinite resource processor while satisfying all the dependence edges to previously scheduled successors. An operation's late time is set to infinity if no successors of the operation have been scheduled. Note that the early and late times of an operation change as scheduling progresses; the interval [*early time*, *late time*] thus represents the range of cycles in which an operation may be scheduled without violating dependencies with currently scheduled operations.

Since backtracking schedulers do not always schedule operations in dependence order, it is possible that the interval [*early time*, *late time*] is empty. Furthermore, even when a range of cycles is available, the operation may have resource conflicts that prevent it from being scheduled within this range. In this case, we undo previous

scheduling decisions. Namely, a backtracking scheduler is said to *forcibly schedule* an operation *x* if it unschedules currently scheduled operations that have resource or dependence conflicts with *x* (conflicting operations), and then schedules *x*. A scheduled operation is *unscheduled* by removing its association with a particular schedule cycle, releasing resources it has reserved, and putting it back in the set of operations that are not yet scheduled. A

```
(1)  while (the list of unscheduled operations is not empty)
(2)    Op = HighestPriorityOperation()   // Op may not be ready
(3)    Compute Op.EarlyTime and Op.LateTime

(4)    success = false
(5)    for (cycle from Op.EarlyTime through Op.LateTime)
(6)      if (resources required by Op are available)
(7)        Schedule(Op, cycle)   // no unscheduling
(8)        success = true
(9)        break
(10)   endfor

(11)   if (success != true)
(12)     ForciblySchedule(Op, ForceCycle(Op))   // Op unschedules its successor(s)
(13) endwhile
```

Figure 2: Main loop of the basic backtracking scheduling algorithm

backtracking scheduler *normally schedules* an operation *x* if it schedules *x* at a cycle in [*early time*, *late time*] without needing to unschedule other conflicting operation(s).

The main loop of a basic backtracking scheduling algorithm, BasicBT, is shown in Figure 2. Each iteration begins by picking the highest priority not yet scheduled operation *Op* (line 2). We compute the *EarlyTime* and *LateTime* of *Op* and then attempt to normally schedule the operation in each cycle from *EarlyTime* to *LateTime* (line 7). If *Op* is not scheduled after iterating through the cycles up to and including *LateTime,* we forcibly schedule it in the cycle determined by the procedure ForceCycle(*Op*) (line 12). In the basic backtracking algorithm, ForceCycle(*Op*) returns the earliest cycle in which *Op* can be scheduled by unscheduling only conflicting successors of *Op* (no predecessors of *Op* or unrelated operations are unscheduled).

The forcible scheduling mechanism ensures that once we select an operation, we are always able to schedule it, even if unscheduling other operations is necessary. The following lemma demonstrates that the basic backtracking algorithm terminates.

Lemma 1

The basic backtracking scheduling algorithm terminates.

Proof:

Consider the while-loop iteration *i* of the backtracking algorithm when we first attempt to schedule an arbitrary operation *x*. It follows that *x*, for this iteration, is the highest priority not yet scheduled operation. Let *y* be the next highest priority not yet scheduled operation, if it exists, and let *S* denote the set of operations that are currently scheduled. Note that the highest priority operation in each iteration is always scheduled. Set *S* therefore includes all the operations in the dependence graph with higher priority than *x*. We will show that after a finite number of iterations, which may include unscheduling and scheduling of operations in *S*, operation *x* and all the operations in *S* are scheduled. That is, after a finite number of iterations the number of scheduled operations increases by one.

There are two ways to schedule operation *x* during iteration *i*. First, if there exists a cycle in [*x.EarlyTime*, *x.LateTime*] for which all the resources required for the execution of *x* are available, the algorithm schedules *x* normally in the earliest such cycle. Note that if *x* has no scheduled successor, *x*'s *LateTime* is set to infinity and we are always able to schedule it without unscheduling other operations.

If *x* cannot be scheduled normally, then the scheduling of *x* is restricted by one or more currently scheduled successors of *x*. Let $G(x)$ be the subgraph of the dependence graph induced by *x* and successors of *x* in *S*. We now consider the operations scheduled and unscheduled from iteration *i* until the iteration that operation *y* is scheduled for the first time. Since an operation can only unschedule its successors, during these iterations only operations in $G(x)$ are scheduled and unscheduled. The number of times an operation in $G(x)$ is unscheduled cannot be larger than the number of times that all its predecessors in $G(x)$ are scheduled. Since operation *x* has no predecessors in $G(x)$, it is scheduled exactly once before *y* is first scheduled. Therefore, there is a finite bound on the number of times any operation in $G(x)$ is scheduled before *y* is scheduled for the first time. We have thus shown that the algorithm schedules operation *x* and all the operations in *S* after a finite number of iterations.

This argument can be reapplied to operation *y*, the operation with the next highest priority after *x*. The same argument applies if operation *x* is the lowest priority operation, i.e., if operation *y* does not exist. Since the number of operations in the dependence graph is finite, the backtracking algorithm terminates.

3.2 The OperBT scheduler

The full-backtracking OperBT scheduler is an aggressive implementation of the basic backtracking algorithm. To improve schedule quality, we allow an operation to unschedule not only its successors but also unrelated operations. In addition to normal scheduling, an operation can unschedule unrelated lower priority operations within its [*EarlyTime*, *LateTime*] interval, and unrelated higher priority operations after its *LateTime*. This allows the

```
(1)  while (the list of unscheduled operations is not empty)
(2)    Op = HighestPriorityOperation()   // Op may not be ready
(3)    Compute Op.EarlyTime and Op.LateTime

(4)    success = false
(5)    for (cycle from Op.EarlyTime through Op.LateTime)
(6)      if (resources required by Op are available)
(7)        Schedule(Op, cycle)   // no unscheduling
(8)        success = true
(9)        break
(10)     elseif (cycle > Op.AttemptedCycle  AND HasHigherPriority(Op, cycle))
(11)       ForciblySchedule(Op, cycle)   // Op unschedules only lower priority operations
(12)       success = true
(13)       break
(14)   endfor

(15)   if (success != true)
(16)     ForciblySchedule(Op, Op.AttemptedCycle + 1)   // Op may unschedule a higher priority operation
(17) endwhile
```

Figure 3: Main loop of the OperBT backtracking scheduler

scheduler to more aggressively schedule important operations as early as possible. To avoid repeatedly scheduling and unscheduling an operation in the same cycle, the OperBT scheduler maintains the *AttemptedCycle* for each operation, the last cycle where it was forcibly scheduled, and if we forcibly schedule the operation again, we do it in *AttempedCycle* + 1.

We show the pseudocode of the OperBT scheduler in Figure 3. It differs from the basic backtracking algorithm in two places. First, the code in lines 10 through 13 allows *Op* to be forcibly scheduled at a cycle in [*EarlyTime*, *LateTime*] if *Op* has a higher priority than a conflicting operation and if *cycle* > *Op.AttempedCycle*. Second, in line 16 the procedure call ForceCycle(*Op*) is instantiated by the expression *Op.AttempedCycle* + 1. That is, if *Op* was last forcibly scheduled at *AttemptedCycle*, then if the operation cannot be scheduled in its [*EarlyTime*, *LateTime*] interval we forcibly schedule it at *AttemptedCycle* + 1, even if this requires unscheduling a higher priority operation. If an operation is forcibly scheduled for the first time, we schedule it at its *EarlyTime*.

Experimental results given in Section 4.1 indicate that the OperBT scheduler is very effective in filling the delay slots of branches. However, the number of unscheduling steps can be excessive. We thus develop a modified backtracking scheduler that reduces the compilation time.

3.3 The ListBT scheduler

The ListBT backtracking scheduler selectively enables backtracking to control the amount of unscheduling, while still maintaining the quality of the overall schedule. Given the objective of filling branch delay slots, only operations with negative incoming latencies are allowed to unschedule other operations. For our machine models, branches have negative incoming edge latencies, and therefore branches can be forcibly scheduled. On the other hand, if our objective were to handle write port resource conflicts between certain high-latency and low-latency operations, we would allow backtracking for selected low-latency operations.

The ListBT scheduler enables backtracking for branches as well as for operations that have been previously unscheduled. In addition, as in the conventional Cycle scheduler, but unlike OperBT, the ListBT scheduler further reduces the number of unscheduling steps by only scheduling ready operations.

The main loop of the ListBT scheduler is shown in Figure 4. The ListBT scheduler selects the highest priority ready operation *Op* (line 2) and attempts to schedule it in each cycle from *EarlyTime* to *LateTime*. *Op* is scheduled if resources are available (line 7), or if backtracking is enabled for *Op* and all conflicting operations have lower priorities (line 11). As in OperBT, if *Op* is not scheduled after iterating through the cycles up to and including *LateTime,* we forcibly schedule it in *Op.AttempedCycle* + 1 (line 17).

```
(1)  while (ReadyList is not empty)
(2)    Op = HighestPriorityOperation(ReadyList)
(3)    Compute Op.EarlyTime and Op.LateTime

(4)    success = false
(5)    for (cycle from Op.EarlyTime through Op.LateTime)
(6)      if (resources required by Op are available)
(7)        Schedule(Op, cycle)   // no unscheduling
(8)        success = true
(9)        break
(10)     elseif (backtracking enabled for Op  AND cycle > Op.AttemptedCycle AND HasHigherPriority(Op, cycle))
(11)       ForciblySchedule(Op, cycle)   // Op unschedules only lower priority operations
(12)       Enable backtracking for unscheduled operations
(13)       success = true
(14)       break
(15)   endfor

(16)   if (success != true)
(17)     ForciblySchedule(Op, Op.AttemptedCycle + 1)   // Op may unschedule a higher priority operation
(18)     Enable backtracking for unscheduled operations
(19) endwhile
```


Figure 4: Main loop of the ListBT backtracking scheduler

At the beginning, backtracking is enabled for branch operations only. However, if an operation is unscheduled by a branch, then the operation may have no valid cycles for scheduling between its *EarlyTime* and *LateTime*. We prevent deadlock by enabling backtracking for any unscheduled operation (lines 12 and 18).

Scheduling step 1	Scheduling step 2	Scheduling step 3	Scheduling step 4	Scheduling step 5	Scheduling step 6
ReadyList ={cmpeq, add}	*ReadyList* = {add}	*ReadyList* = {load}	*ReadyList* = {bfalse}	*ReadyList* = {add}	*ReadyList* = {load}
0: **cmpeq**	0: cmpeq 1: **add**	0: cmpeq 1: add 2: **load**	0: cmpeq 1: *add* <-- **bfalse** 2: load	0: cmpeq 1: bfalse 2: *load* <-- **add**	0: cmpeq 1: bfalse 2: add 3: **load**
cmpeq normally scheduled	add normally scheduled	load normally scheduled	bfalse forcibly sched., add unscheduled	add forcibly sched., load unscheduled	load normally sched.

Figure 5: Operation of the ListBT scheduler

In Figure 5 we show the scheduling steps of the ListBT scheduler when applied to the example in Figure 1. In this case ListBT finds an optimal schedule. Note that ListBT schedules the first three operations without backtracking. Since `bfalse` has higher priority than the `add` operation, in step 4 it is forcibly scheduled at its *EarlyTime*, cycle 1. The `add` operation is unscheduled and returned to the *ReadyList*, and backtracking is enabled for it. In step 5 the `add` then unschedules the lower priority `load` operation which is finally scheduled in cycle 3.

We have found that in practice the OperBT and ListBT schedulers converge quickly. However, showing that the schedulers terminate is an open question.

4 Experimental evaluation

In this section we experimentally evaluate the Cycle, BasicBT, OperBT, and ListBT schedulers. We use a variety of measures, including number of empty delay slots, optimally scheduled regions, dynamic cycles, to show that opportunities exist for backtracking schedulers to produce significantly better schedules. We use the runtime of each scheduler to show that backtracking schedulers can indeed be used in production compilers. Since the performance of BasicBT is worse than the more sophisticated OperBT and ListBT schedulers, for BasicBT we only report dynamic cycles.

We apply the schedulers to seven benchmarks from the SPECint95 suite: *compress, go, ijpeg, li, m88ksim, perl*, and *vortex*. The *gcc* benchmark is not successfully compiled by the front end of the Trimaran compiler and is not

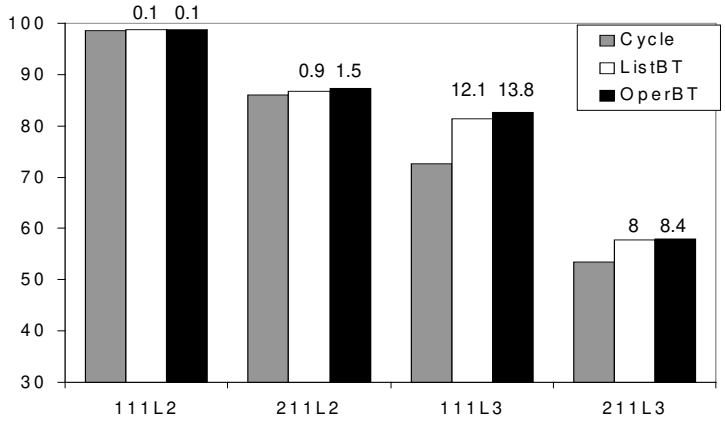


Figure 6: Percent of filled branch delay slots by the Cycle, ListBT, and OperBT schedulers for four processors (geometric mean over seven SPECint95 benchmarks). The numbers above the bars are the percent increase of ListBT and OperBT over Cycle.

included in our experiments. Our research targets general-purpose integer applications that are well represented by these benchmarks. We evaluate the schedulers on six processors: 111L1, 211L1, 111L2, 211L2, 111L3, and 211L3 (shown in Table 1), whose characteristics make them representative of a wider class of processors. For each measure and processor we report the geometric mean over the seven benchmarks.

Our experimental setup consists of the Trimaran version 2.00 running on Linux. This distribution includes version 2.33 of the IMPACT compiler. The compiler runs on a 4x450MHz Intel Pentium II Xeon with 2 GB of RAM.

4.1 Branch delay slots and optimally scheduled regions

A branch has an empty delay slot if there is an empty instruction (instructions that contains no operations) no more than branch latency-1 cycles later. A branch delay slot is filled if it is not empty. We count branch delay slots statically after all scheduling phases have completed. We show in Figure 6 the percentage of branch delay slots that are filled by each scheduler and the percentage improvement over the Cycle scheduler. Over all processors and benchmarks the Cycle scheduler leaves 24.2% of all delay slots empty. Increasing the branch latency from 2 to 3 causes the fraction of empty delay slots to increase from a geometric mean of 7.9% to 37.6%, more than a fourfold increase. These results indicate that as the branch latency increases, it may be possible to significantly improve schedules generated by the Cycle scheduler. With a branch latency of 3, the OperBT scheduler fills 69.2% of all delay slots, reducing the number of unfilled delay slots from 37.6% to 30.8% of all delay slots. The ListBT scheduler performs nearly as well, filling 68.6% of all delay slots.

The number of filled branch delay slots is a measure of how well the scheduler is exploiting the processor's available parallelism. A more direct measure of schedule quality is the number of optimally scheduled branches and

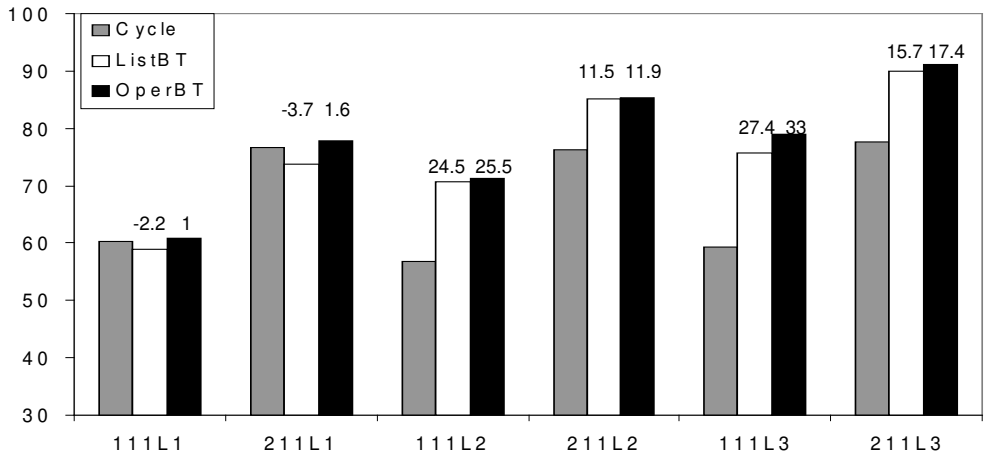


Figure 7: Percent of superblocks optimally scheduled by the Cycle, ListBT, and OperBT schedulers for six processors (geometric mean over seven SPECint95 benchmarks). The numbers above the bars are the percent increase of ListBT and OperBT over Cycle.

superblocks. We compute early time bounds using tight, dependence- and resource-based bounds[4-6]. A branch is optimally scheduled if its schedule cycle equals its early time bound. A superblock is optimally scheduled if every branch within it is optimally scheduled.

We show in Figure 7 that for non-unit branch latency the OperBT scheduler increases the percentage of superblocks optimally scheduled over the Cycle scheduler from a geometric mean of 66.9% to 81.4%, an increase of 21.7%. For non-unit branch latency the ListBT scheduler optimally schedules 80% of superblocks, an increase of 19.6% over the Cycle. Note that the actual fraction of superblocks optimally scheduled may be even higher since the bounds may differ from the optimal values.

4.2 Dynamic Cycles

We compute a tight lower bound on the number of dynamic cycles consumed by any benchmark by taking the sum over all branches of the branch's early time bound plus its latency, times its weight (number of times the branch is taken during profiling). The gap between this bound and the number of dynamic cycles in a schedule produced by the Cycle scheduler represents the size of the potential improvement. We show in Table 2 that the size of this gap increases from 3.4% to 4.8% as the branch latency increases from 1 to 3. The table also gives the maximum values over the seven benchmarks. For the branch latency of 2 the OperBT and ListBT schedulers produce schedules that are 1.1% better in terms of dynamic cycles than schedules produced by the Cycle scheduler, and for the branch latency of 3 the speedup increases to 1.9%. For comparison, Table 2 also includes the data for the basic backtracking

Table 2: Percent speedup of the BasicBT, ListBT, and OperBT backtracking schedulers over the Cycle scheduler for six processors (geometric mean and maximum value over seven SPECint95 benchmarks)

Processor	BasicBT (max)	ListBT (max)	OperBT (max)	Bound (max)
111L1	0.0 (0.2)	-0.1 (0.0)	0.0 (0.2)	3.4 (5.6)
211L1	0.6 (1.7)	-0.2 (0.1)	0.6 (1.7)	4.4 (6.8)
111L2	0.4 (0.7)	1.1 (1.7)	0.5 (1.7)	3.7 (5.8)
211L2	1.6 (3.1)	1.3 (3.2)	1.7 (3.2)	4.7 (7.2)
111L3	0.6 (1.1)	1.6 (2.5)	1.2 (1.9)	4.1 (6.2)
211L3	2.7 (7.3)	2.5 (7.4)	2.8 (7.6)	4.8 (9.7)

scheduling algorithm, BasicBT, from Section 3.1. As expected, its performance is always better than the Cycle scheduler and in general worse than the more aggressively backtracking OperBT and ListBT schedulers.

Results show that backtracking schedulers are more efficient on wider than on narrow processors. On a wider processor it is easier for the backtracking schedulers to move operations and fill delay slots. This added flexibility accounts for the increase in performance. We also note that OperBT outperforms ListBT for the 111L3, while the opposite is true for 211L3.

4.3 Scheduling steps and runtime

As earlier described, backtracking schedulers schedule operations normally and forcibly. The conventional Cycle scheduler schedules operations only normally. We estimate the amount of work done by each scheduler by counting the total number of times operations are scheduled, i.e., the total number of while-loop iterations (scheduling steps). This data is shown in Table 3. Note that for unit branch latency, the number of scheduling steps is the same for the three schedulers; the weighted priority function, described in Section 3.1, guarantees that no operations need to be forcibly scheduled. For the narrow 111 processor with non-unit branch latency, the OperBT scheduler uses 49% more scheduling steps than the Cycle scheduler, while for the wider 211 processor this number decreases to 9% because of the greater availability of scheduling slots. For non-unit branch latency the ListBT scheduler uses about 15% fewer scheduling steps than OperBT.

The number of forcible scheduling steps measures the extra, backtracking work done by the OperBT and ListBT schedulers. As noted above there is no backtracking for unit branch latency. The ListBT scheduler uses five times fewer forcible scheduling steps than the OperBT scheduler for a branch latency of 2, and about three times fewer for a latency of 3.

Table 3: Ratio of scheduling steps and runtime of the ListBT and OperBT backtracking schedulers versus the Cycle scheduler for six processors (geometric mean and maximum value over seven SPECint95 benchmarks)

Processor	Scheduling steps (max)		Runtime (max)	
	ListBT	OperBT	ListBT	OperBT
111L1	1.00 (1.00)	1.00 (1.00)	0.86 (0.92)	1.03 (1.04)
211L1	1.00 (1.00)	1.00 (1.00)	0.94 (0.97)	1.02 (1.03)
111L2	1.05 (1.09)	1.56 (2.90)	0.88 (0.93)	1.26 (1.71)
211L2	1.01 (1.03)	1.11 (1.42)	0.95 (0.98)	1.04 (1.11)
111L3	1.07 (1.14)	1.43 (2.42)	0.89 (0.94)	1.20 (1.56)
211L3	1.02 (1.02)	1.07 (1.23)	0.95 (0.97)	1.04 (1.09)

We also measure the runtime of the three schedulers through calls to Trimaran timing routines that use the clock() library function. The ratios shown in the right half of Table 3 reflect the superblock scheduling routines only, the phase that consumes about 30% of the overall backend compilation time. As expected, the OperBT scheduler is the most time consuming, taking up to 26% longer than the Cycle scheduler. However, the ListBT scheduler outperforms OperBT and even Cycle. The Cycle scheduler schedules operations cycle by cycle, and repeatedly attempts to schedule high priority operations, even if resource constraints force them to be scheduled later. Runtime results indicate that ListBT and even OperBT are fast enough to be used in production compilers.

5 Related Work

The early work on parallel instruction scheduling for processors was carried out in the context of microcode compaction [14, 15]. Heuristics for scheduling instructions in a basic block for pipelined processors [3, 16] and superscalar processors [4] typically rely on ordering the operations in a block based on priority functions and scheduling the operations one by one as in Cycle scheduling. Approaches for basic block scheduling using optimization techniques such as finite state automata have been studied [17].

The amount of parallelism within a basic block is limited and not sufficient for modern EPIC processors. Global schedulers use a larger scheduling region and perform code motion between basic blocks. The trace scheduler constructs a trace consisting of a linear chain of basic blocks with multiple entries and exits [18, 19]. Global schedulers restrict the scheduling regions to reduce the complexity of inserting compensation code in the side entries/exits due to code motion [20-22]. The schedulers described in this paper use the superblock [12, 23] and

hyperblock [24] as the global scheduling region. Wavefront scheduling uses a region consisting of a tree of basic blocks [25]. Meld scheduling extends the scope of the optimization without increasing the complexity associated with code motion by allowing latency constraints to propagate outside the scheduling [26].

Schedulers may incorporate various levels of backtracking to improve the quality of schedules. The Multiflow trace scheduling compiler [19] employs a limited form of backtracking for memory operations only. It allows an operation to be unscheduled at most once and it does not refill the issue slot of the unscheduled operation. Wang et al. [27] consider backtracking for the problem of scheduling precedence-constrained tasks on parallel processors. Their algorithm uses a single level of backtracking; it allows immediate predecessor rescheduling, in certain circumstances, to find better schedules with only a small increase in processing times. Hoogerbrugge and Augusteijn [28] describe a simple backtracking strategy to fill branch delay slots when scheduling for Philips Trimedia processor. Their scheduler ignores the control dependences from non-branch operations to a branch *b* and schedule *b* as early as possible. If it turns out that *b* was scheduled too early, the scheduler unschedules the branch as well as the operations scheduled after it and attempts to schedule *b* one cycle later. To reduce the amount of backtracking, a lower bound on the cycle where a branch can be legally scheduled is computed. In contrast, our ListBT and especially OperBT schedulers apply unrestricted, aggressive backtracking policies.

Programs spend a significant fraction of the total execution time in loops and special scheduling techniques have been developed for loops [5, 29, 30]. Rau developed the iterative modulo scheduler that backtracks in a manner similar to the ListBT scheduler [10]. A modulo scheduler attempts to schedule for a particular initiation interval (II) and typically increases the II if it is unable to generate a schedule within the prescribed compile time budget. All schedules that meet a particular II are essentially equivalent. These unique characteristics of modulo scheduling give rise to a different set of algorithmic choices. First, the iterative modulo scheduler may occasionally get locked in a repetitive orbit. However, once the compile time budget is exhausted, the iterative modulo scheduler attempts to schedule using a different II. The OperBT and ListBT schedulers are guaranteed to never revisit the same partial schedule. This guarantee is essential because there is no option similar to increasing the II and trying to schedule again. Second, the iterative modulo scheduler must account for a cyclic dependence graph, which may constrain the operation late time even when no successor from the same loop iteration has been scheduled. The OperBT and

ListBT schedulers do not face such constraints. Third, the schedule length is the primary optimization metric in acyclic scheduling as opposed to being a secondary metric in modulo scheduling.

6 Conclusions

This paper motivates the need for backtracking schedulers by describing processor features such as branch conflicts and resource conflicts that cannot be addressed adequately by conventional, non-backtracking, schedulers. We have presented the backtracking approach for acyclic instruction scheduling. We have described a generic backtracking scheduler and proved that it terminates. The more aggressive backtracking scheduler, OperBT, picks operations in priority order and permits any operation to unschedule already scheduled operations. The ListBT selective backtracking scheduler picks operations primarily in dependence order and therefore greatly reduces the need for backtracking. Experiments have demonstrated that both backtracking schedulers successfully fill a significant fraction of branch delay slots, providing an overall performance increase of between 1-7%. We conclude that with a small amount of additional computation, aggressive backtracking-based schedulers can effectively improve schedule quality by filling branch delay slots.

Acknowledgements

The ListBT scheduler arose from suggestions by Scott Mahlke and Bob Rau. The Elcor scheduling framework was architected in conjunction with Vinod Kathail and implemented with assistance from Brian Dietrich. The authors thank all the members of the Compiler and Architecture Research (CAR) group at Hewlett-Packard Laboratories who were engaged in the Elcor compiler project.

References

1. Schlansker, M.S. and B.R. Rau, *EPIC: Explicitly parallel instruction computing.* Computer, 2000. **33**(2): p. 37-45.
2. Muchnick, S., *Advanced Compiler Design and Implementation*. 1997: Morgan Kaufmann.
3. Gibbons, P.B. and S.S. Muchnick. *Efficient instruction scheduling for a pipelined architecture*. in *ACM SIGPLAN Symposium on Compiler Construction*. 1986.
4. Warren, H.S., *Instruction scheduling for the IBM RISC System/6000 processor.* IBM Journal of Research and Development, 1990. **34**(1): p. 85-92.
5. Dehnert, J.C. and R.A. Toole, *Compiling for the Cydra-5.* Journal of Supercomputing, 1993. **7**(1/2): p. 181-227.
6. Rim, M. and R. Jain, *Lower-bound performance estimation for the high-level synthesis scheduling problem.* IEEE Transactions on CAD of ICS, 1994. **13**(4): p. 452-459.
7. Langevin, M. and E. Cherny, *A recursive technique for computing lower-bound performance of schedules.* ACM Transactions on Design Automaton of Electronic Systems, 1996. **1**(4): p. 443-455.

8. Eichenberger, A. and W.M. Meleis. *Balance Scheduling: Weighting branch tradeoffs in superblocks*. in *International Symposium on Microachitecture*. 1999. Haifa, Israel.
9. Abraham, S.G., *Efficient backtracking instruction schedulers*. 2000, Hewlett-Packard Laboratories.
10. Rau, B.R., *Iterative modulo scheduling.* International Journal of Parallel Programming, 1996. **24**(1): p. 3-64.
11. Kathail, V., M.S. Schlansker, and B.R. Rau, *HPL PlayDoh Architecture Specification: Version 1.0*. 1991, Hewlett-Packard Laboratories.
12. Hwu, W.W., S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, *The superblock: An effective technique for VLIW and superscalar compilation.* Journal of Supercomputing, 1993. **7**(1/2): p. 229-248.
13. Fisher, J.A., *Global code generation for instruction-level parallelism: Trace scheduling-2*. 1993, Hewlett-Packard Laboratories.
14. Davidson, S., D. Landskov, B.D. Shriver, and P.W. Mallet, *Some experiments in local microcode compaction for horizontal machines.* IEEE Transactions on Computers, 1981. **C-30**(7): p. 460-477.
15. Rau, B.R. and J.A. Fisher, *Instruction-level parallel processing: History, overview and perspective.* Journal of Supercomputing, 1993. **7**(1/2): p. 9-50.
16. Hennessy, J.L. and T. Gross, *Postpass code optimization of pipeline constraints.* ACM Transactions on Programming Languages and Systems, 1983. **5**(3): p. 422-448.
17. Bala, V. and N. Rubin, *Efficient instruction scheduling using finite state automata.* International Journal of Parallel Programming, 1997. **25**(2): p. 53-82.
18. Fisher, J., *Trace scheduling: A technique for global microcode compaction.* IEEE Transactions on Computers, 1981. **C-30**(7): p. 478-490.
19. Lowney, P.G., S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, *The Multiflow trace scheduling compiler.* Journal of Supercomputing, 1993. **7**(1/2): p. 51-142.
20. Ferrante, J., K.J. Ottenstein, and J.D. Warren, *The program dependence graph and its use in optimization.* ACM Transactions on Programming Languages and Systems, 1987. **9**(3): p. 319-349.
21. Bernstein, D. and M. Rodeh. *Global instruction scheduling for superscalar processor*. in *ACM SIGPLAN Symposium on Programming Languages Design and Implementation*. 1991. Toronto, Canada.
22. Moon, S.M. and K. Ebcioglu. *An efficient resource-constrained global scheduling method for superscalar and VLIW processors*. in *International Symposium on Microachitecture*. 1992. Portland, Oregon.
23. Chang, P.P., S.A. Mahlke, and W.W. Hwu, *Using profile information to assist classic code optimizations.* Software Practice and Experience, 1991. **21**(12): p. 1301-1321.
24. Mahlke, S.A., D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. *Effective compiler support for predicated execution using the hyperblock*. in *International Symposium on Microachitecture*. 1992. Portland, Oregon.
25. Bharadwaj, J., K. Menezes, and C. McKinsey. *Wavefront scheduling: Path based data representation and scheduling of subgraphs*. in *International Symposium on Microarchitecture*. 1999. Haifa, Israel.
26. Abraham, S.G., V. Kathail, and B.L. Dietrich, *Meld scheduling: A technique for relaxing scheduling constraints.* International Journal of Parallel Programming, 1998. **26**(4): p. 349-381.
27. Wang, Y., N. Amato, and D. Friesen. *Hindsight helps: Deterministic task scheduling with backtracking*. in *International Conference on Parallel Processing*. 1997. Bloomington, Illinois.
28. Hoogerbrugge, J. and L. Augusteijn, *Instruction scheduling for TriMedia.* Journal of Instruction-Level Parallelism, 1999(1).
29. Charlesworth, A.E., *An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family.* Computer, 1981. **14**(9): p. 18-27.
30. Rau, B.R., D.W.L. Yen, W. Yen, and R.A. Towle, *The Cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs.* Computer, 1989. **22**(1): p. 12-34.