Efficient Backtracking Instruction Schedulers

Santosh G. Abraham
Hewlett-Packard Laboratories
Palo Alto, CA 94304
abraham@hpl.hp.com

Waleed M. Meleis and Ivan D. Baev
Electrical and Computer Engineering Dept.
Northeastern University, Boston, MA 02115
{meleis, baev}@ece.neu.edu

Abstract

Current schedulers for acyclic regions schedule operations in dependence order and never undo a scheduling decision. In contrast, backtracking schedulers may unschedule operations and can often generate better schedules.

In this paper, we describe a conventional cycle scheduler followed by two novel backtracking schedulers, OperBT and ListBT. The full-backtracking OperBT scheduler enables backtracking for all operations and unschedules operations to make space for the current operation. The OperBT scheduler increases the percentage of superblocks scheduled optimally over a conventional non-backtracking scheduler from an average of 66.9% to 81.4%, an increase of 21.7%.

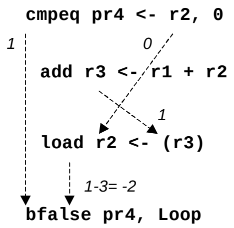
The selective backtracking ListBT scheduler enables backtracking only when scheduling operations for which backtracking is likely to be advantageous. This hybrid scheduler is almost as good as the OperBT backtracking scheduler in terms of generated code quality, but backtracks about four times less often.

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. 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. 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 schedules operations one by one, and does not undo or revisit scheduling decisions that were made previously. This paper develops and evaluates backtracking schedulers that sometimes 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. Simple RISC processors may have five pipe stages, and 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. 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. Consider an arithmetic operation A generating a live-out value, and a succeeding branch B. The latency of the dependence edge from A to B must guarantee that the branch does not transfer control to another scheduling region before A generates its live-out value. If the branch latency exceeds the arithmetic operation latency, this edge latency is negative, permitting A to descend below B into branch delay slots.

As demonstrated in the example in Figure 1, current schedulers that schedule instructions cycle by cycle are not effective in handling such negative latencies and filling delay slots. 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.



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

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. The example also demonstrates a backtracking scheduler. Here the `bfalse` has a higher priority than the `add` and displaces the scheduled `add`, that 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 generally schedule operations in dependence order.

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 memories accessed frequently is high and can account for a large fraction of total on-chip power consumption. Accordingly, exposing the branch latency and reducing/eliminating prediction hardware may be an attractive alternative for mobile processor designs.

The third trend is toward wide-issue processors. Media-intensive applications have large amounts of parallelism that can be effectively exploited by processors that issue many operations in each cycle [2]. However, wide-issue processors also require a commensurate increase in the number of register read and write ports. Since increasing the number of write ports is especially difficult and expensive, 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, a method that works locally across a few instructions. However peephole optimization strategies tend to work well only when they are developed specifically for a particular design with its own 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. Another alternative is a bottom-up scheduler that first schedules the last branch, and then proceeds to schedule predecessors of the branch as late as possible. While this approach will tend to fill delay slots associated with the last branch, it does not do a good job of filling delay slots of earlier branches.

Table 1. Processor configurations

Processor config.	Integer units	Floating -point units	Load/ store units	Bra-nch latency	Max. 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

This paper first describes the overall scheduling model consisting of a processor model, scheduler input and output. Then we present the operation of a conventional **Cycle** scheduler that schedules operations cycle by cycle or in VLIW parlance, instruction by instruction.

We demonstrate that the **Cycle** scheduler cannot fill branch delay slots effectively. The **OperBT** scheduler is a full backtracking scheduler that attempts to schedule operations in priority order. This scheduler fills branch delay slots successfully but may unschedule operations repeatedly. The **ListBT** scheduler is a selective backtracking scheduler that schedules operations in dependence order and selectively backtracks when it is likely to be profitable. This scheduler is almost as effective in filling branch delay slots but has better compile times than the full backtracking scheduler. We present a detailed evaluation of all three schedulers on a set of SPECint95 benchmarks.

2 Scheduling Model

2.1 Processor architecture

We use a family of VLIW processors based on the HPL-PD architecture [3]. 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 [4], 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 of the individual exits of the basic blocks. 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 exit 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.
2. resource constraints are satisfied, i.e. in our simplified processors, the number of a certain type of operation 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 exits. The contribution of a particular exit is the product of the number of times this exit was taken during profiling and its exit time. The exit time of a branch is the sum of the branch's issue cycle and the branch latency.

3 Conventional schedulers

This section describes common pre-scheduling steps as well as a conventional scheduler that does not backtrack. Guided by a priority list, the **Cycle** scheduler schedules all the operations to be issued in a particular cycle before going on to the next cycle.

3.1 Common pre-scheduling steps

All schedulers described in this paper start by computing early and late times for each operation. The *early time* of an operation is the earliest time that it can be issued on a processor with infinite resources. The *start operation* is the control-merge at the beginning of the superblock on which all operations are dependent. The length of a path from operation A to operation B is the sum of the latencies of the edges in the path from operation A to operation B. The early time of an operation is the longest path from the start operation to the operation under consideration. The *late time* of an operation A with respect to an exit E is the latest cycle at which operation A can be issued on an infinite resource machine while still issuing exit E at its early time. The late time of an operation A is computed as the early time of operation E minus the longest path from operation A to exit E.

Let *maxheight* be the maximum early time among all operations in the superblock, and *height* of an operation with respect to an exit E be *maxheight* minus the late time of that operation with respect to E. We let *weighted height* of an operation be the sum over all superblock exits, E, of the product of the profiled weight of E and the height of the operation with respect to E. Though the Elcor compiler supports several priority functions, all the evaluations reported here are based on the weighted height priority function [5].

3.2 Conventional Cycle Scheduler

Before we describe the main scheduling loop of the **Cycle** scheduler, we present some concepts and data structures. The *CurrentCycle* is the cycle in which

OperBT scheduler

```
UnschedList = all operations sorted by priority
while (CurrentOperation = UnschedList.pop())
  Compute EarlyCycle and LateCycle for CurrentOperation
  ForceCycle = max(Attempted Cycle+1, EarlyCycle)

  success = FALSE
  for (CurrentCycle from EarlyCycle through LateCycle)
    if (resources required by CurrentOperation available)
      Schedule(CurrentOperation, CurrentCycle)
      success = TRUE; break
    elseif ((CurrentCycle >= ForceCycle) &&
          (HasHigherPriority(CurrentOperation, CurrentCycle))
      Unschedule(conflicting operations, CurrentCycle)
      UnschedList.update()
      Schedule(CurrentOperation, CurrentCycle)
      success = TRUE; break
    endif
  endfor
  if (success == FALSE)
    Unschedule(conflicting operations, ForceCycle)
    UnschedList.update()
    Schedule(CurrentOperation, ForceCycle)
    Set Attempted Cycle = ForceCycle for CurrentOperation
  endif
endwhile
```

ListBT scheduler

```
ReadyList = start operation
while (CurrentOperation = ReadyList.pop())
  Compute EarlyCycle and LateCycle for CurrentOperation
  ForceCycle = max(Attempted Cycle+1, EarlyCycle)

  success = FALSE
  for (CurrentCycle from EarlyCycle through LateCycle)
    if (resources required by CurrentOperation available)
      Schedule(CurrentOperation, CurrentCycle)
      ReadyList.update()
      success = TRUE; break
    elseif ((unscheduling enabled for CurrentOperation) &&
          (CurrentCycle >= ForceCycle) &&
          (HasHigherPriority(CurrentOperation, CurrentCycle))
      Unschedule(conflicting operations, CurrentCycle)
      Enable unscheduling for conflicting operations
      Schedule(CurrentOperation, CurrentCycle)
      ReadyList.update()
      success = TRUE; break
    endif
  endfor
  if (success == FALSE)
    Unschedule(conflicting operations, ForceCycle)
    Enable unscheduling for conflicting operations
    Schedule(CurrentOperation, ForceCycle)
    ReadyList.update()
    Set Attempted Cycle = ForceCycle for CurrentOperation
  endif
endwhile
```

Figure 2. Main scheduling loop for OperBT and ListBT schedulers

operations are being scheduled currently by the scheduler. The *CurrentCycle* is initially set to 0, and incremented when no more operations can be scheduled in that cycle because of dependence or resource constraints. The *CurrentOperation* is the operation currently being considered for scheduling. The *ScheduleCycle* is the issue cycle assigned to an operation by the scheduler.

The *EarlyCycle* is the earliest cycle that an operation can be scheduled. On entering the main scheduling loop, the *EarlyCycle* of an operation is set to its early time. If the operation is found to be not schedulable at its *EarlyCycle*, then its *EarlyCycle* is incremented, so that we do not repeatedly and unsuccessfully attempt to schedule the operation in a particular cycle. A *ready operation* is an operation whose predecessors have been scheduled, and the *ReadyList* is the list of all ready operations. A *ready operation for the CurrentCycle* is a ready operation whose *EarlyCycle* is not more than the *CurrentCycle* and whose latency constraints on its incoming edges will not be violated by scheduling it in the *CurrentCycle*. The *CCReadyList* is the list of all ready operations for the *CurrentCycle*.

The main scheduling loop iterates until all operations have been scheduled. In each iteration we recompute the *CCReadyList*. If the *CCReadyList* is empty, there are no more operations that can be scheduled in the *CurrentCycle*. Therefore, we increment *CurrentCycle* and continue on to the next iteration. If the *CCReadyList* has one or more operations, we remove the highest priority operation from the *CCReadyList* and set *CurrentOperation* equal to it. If the *CurrentOperation* has no resource conflicts with already scheduled operations, we schedule it in the *CurrentCycle*. Otherwise, the *CurrentOperation* cannot be scheduled in the *CurrentCycle* because of resource conflicts. We increment the operation's *EarlyCycle* to ensure that we do not consider it for scheduling again in the *CurrentCycle*. This completes the description of the main scheduling loop. Details about the incremental recomputation of the *ReadyList* and *CCReadyList* can be found in [6].

4 Backtracking schedulers

In this section we describe two novel backtracking schedulers: **OperBT** and **ListBT**.

4.1 Common concepts

Since backtracking schedulers do not always schedule operations in dependence order, it is possible that an operation's predecessor(s) and successor(s) may already be scheduled. As a result, there may only be a limited (or even null) range of cycles in which the *CurrentOperation* may be scheduled without violating dependencies with already scheduled operations or creating resource conflicts.

In such situations we need a mechanism to make forward progress. The backtracking scheduler is said to *forcibly schedule* an operation A if it removes already scheduled operations that have resource or dependence conflicts with A and then schedules A. The *ForceCycle* is the cycle in which the scheduler forcibly schedules an operation. A scheduled operation B is *unscheduled* by removing its association with a particular schedule cycle, releasing resources it may have reserved, putting it back among the pool of operations to be scheduled and in general, undoing any steps that were performed when B was last scheduled. The forcible scheduling mechanism ensures that once we select a *CurrentOperation*, we are always able to successfully schedule it, even if that requires unscheduling other operations.

We must also ensure that the scheduler does not enter an infinite loop in which it, say, unschedules operation A to schedule B and later unschedules operation B to schedule operation A in the same cycle. To avoid such termination problems we maintain for each operation *AttemptedCycle*, the last attempted cycle where we forcibly scheduled that operation. When we first unschedule a particular operation A, we set its *AttemptedCycle* to (*ScheduleCycle*-1). Thereafter, if we forcibly schedule A, we ensure that it is forcibly scheduled at more than its *AttemptedCycle*, and then update the *AttemptedCycle* to *ForceCycle*, the cycle in which A is forcibly scheduled.

Unlike the conventional scheduler, not all predecessors of an operation may be scheduled at the time when the operation is considered for scheduling. Therefore, the *EarlyCycle* of an operation is the maximum of its early time and the earliest time the operation can be issued while satisfying all dependence edges from predecessor scheduled operations. Also, unlike the conventional scheduler, there are bounds on how late an operation can be scheduled. The *LateCycle* of an operation is the latest time the operation can be issued while satisfying all the dependence edges to successor scheduled operations. As in the conventional scheduler, we compute early times, late times, and priorities for each operation before entering the main scheduling loop.

4.2 OperBT scheduler

The **OperBT** scheduler maintains a priority sorted list, *UnschedList*, of unscheduled operations. The main scheduling loop shown in Figure 2 iterates until *UnschedList* is empty. In each iteration we remove the highest priority operation from *UnschedList* and set *CurrentOperation* equal to it. We first compute the *EarlyCycle* and *LateCycle* of the *CurrentOperation* and then attempt to schedule it in each cycle from *EarlyCycle* to *LateCycle*. We schedule the *CurrentOperation*, if resources are available or if the conflicting operation(s) occupying the required resources have lower priority. In the latter case, we unschedule the conflicting operations. If we do not schedule the *CurrentOperation* after iterating through the cycles up to and including *LateCycle*, we set *ForceCycle* to the maximum of its *AttemptedCycle*`+1` and the *EarlyCycle*, forcibly schedule the operation in the *ForceCycle*, and finally set its *AttemptedCycle* to the *ForceCycle*. We prove that the **OperBT** scheduler does not deadlock and does terminate with a complete schedule in [6].

Experimental results indicate that the **OperBT** scheduler is very effective in filling the delay slots of branches. However, the number of unscheduling steps might be excessive. In the next section, we develop a modified backtracking algorithm that normally schedules operations in dependence order to reduce the number of backtracking (unscheduling) steps.

4.3 ListBT Scheduler

The **ListBT** scheduler normally schedules operations in dependence order. Only ready operations, those whose predecessors are scheduled, are considered for scheduling. The *ReadyList* is a priority sorted list of ready operations. The **ListBT** scheduler selectively enables forcible scheduling to control the amount of unscheduling, while still maintaining the quality of the overall schedule. Given the objective of successfully filling branch delay slots, only operations with negative incoming latencies are allowed to unschedule other operations. For our machine models, only branches have negative incoming edge latencies. If the objective is, say, to handle write port resource conflicts between high-latency and low-latency operations, we allow the low-latency operations to unschedule high-latency operations.

Initially the **ListBT** scheduler schedules operations in dependence order; the *LateCycle* of an operation is infinity and the operation can always be successfully scheduled in some cycle. Once an operation B is unscheduled, it may have a finite range of valid cycles between its *EarlyCycle* and *LateCycle*. If unscheduling is disabled for B, we may not be able to successfully schedule it, leading to a deadlock. Therefore, unscheduling is enabled for any operation that is unscheduled for the first time, and for such operation we maintain *AttemptedCycle* and *ForceCycle* as in the **OperBT** scheduler. Similarly, we never forcibly schedule an operation in the same cycle twice. Using this property and other aspects of the scheduling algorithm, we can show that **ListBT** always terminates.

The **ListBT** main scheduling loop, whose pseudocode is shown in Figure 2, parallels that of the **OperBT** scheduler. (On the implementation level, the main scheduling loop of all the three schedulers described in this paper is expressed by the same unit of C++ code.) The *ReadyList*, initially set to the start operation of the

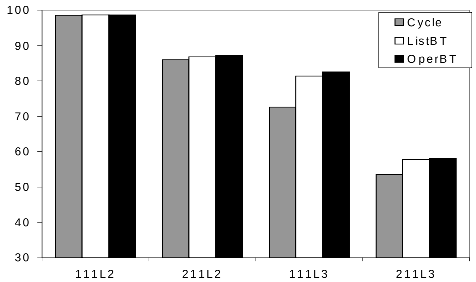


Figure 3. Percent of filled branch delay slots

superblock, is the structure over which the **ListBT** scheduling loop iterates.

5 Experimental evaluation

In this section we experimentally evaluate the quality of the schedules produced by the **Cycle**, **OperBT**, and **ListBT** schedulers. We applied the schedulers to seven benchmarks from the SPECint95 suite (the most recent benchmark certified on Trimaran): *compress, go, ijpeg, li, m88ksim, perl*, and *vortex*. Our research targets general purpose integer applications which 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.

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.

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 3 the percentage of branch delay slots that are filled by each. 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

Table 2. Percent speedup over the Cycle scheduler

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

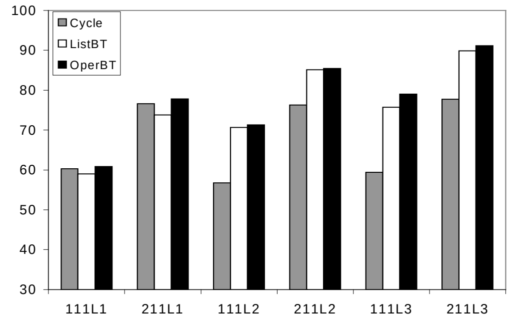


Figure 4. Percent of superblocks optimally scheduled

fraction of empty delay slots to increase from an average of 7.9% to 37.6%, more than a fourfold increase. With a branch latency of three, 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 superblocks. We compute early time bounds using the tight, dependence and resource-based bounds, introduced by Rim and Jain [7] and Langevin and Cerny [8] and applied to superblock scheduling in Eichenberger and Meleis [9]. A branch is scheduled optimally if its issue time equals its early time bound. A superblock is scheduled optimally if every branch within it is scheduled optimally.

We show in Figure 4 that for non-unit branch latency. the **OperBT** scheduler increases the percentage of superblocks scheduled optimally over the **Cycle** scheduler from an average of 66.9% to 81.4%, an increase of 21.7%. For non-unit branch latency, the **ListBT** scheduler schedules optimally 80% of superblocks, an increase of 19.6% over **Cycle**.

5.1 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

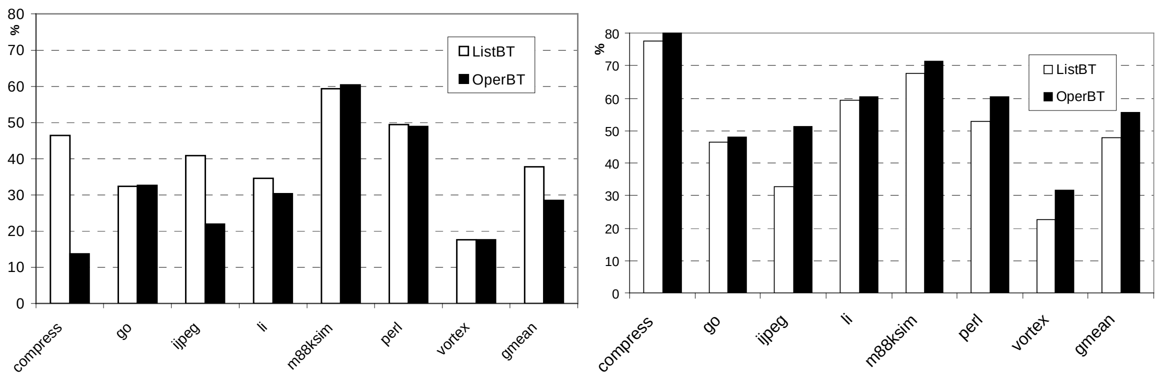


Figure 5. Percent of Cycle gap eliminated by backtracking schedulers for 111L3 and 211L3

latency increases from 1 to 3. The table also gives the maximum values over the six benchmarks. For the branch latency of two, the backtracking schedulers produce schedules that are on average 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%.

Given the gap between the bound and schedules produced by the **Cycle** scheduler, we measure the fraction of this performance gap that is eliminated by the backtracking schedulers. This data is shown in Figure 5. For a branch latency of 3 the **ListBT** scheduler reduces the performance gap by 42.8%; i.e. more than 42.8% of the performance lost by the **Cycle** scheduler has been regained. Similarly, the **OperBT** scheduler regains more than 42.1% of the performance lost by the **Cycle** scheduler, again with a branch latency of 3.

5.2 Compile time and scheduling steps

In Table 3 we estimate the amount of work done by each scheduler by counting the total number of times operations are scheduled. For the narrow 111 processor with non-unit branch latency, the **OperBT** scheduler uses 50% 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, on average the **ListBT** scheduler uses about 20% fewer scheduling steps than **OperBT**.

The number of forcible scheduling steps measures the extra, backtracking work done by the **OperBT** and **ListBT** schedulers. On average 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.

We also measure the runtime of the three schedulers using calls to Trimaran timing routines that use the clock() library function. The data shown in the right half of Table 3 gives the runtime of the superblock scheduling routines only. 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**.

6 Related Work

Heuristics for scheduling instructions in a basic block for pipelined processors [10] and superscalar processors [11] typically rely on ordering the operations in a block based on priority functions and scheduling the operations one by one as in **Cycle** scheduling. 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 [5, 12]. Global schedulers restrict the scheduling regions to reduce the complexity of inserting compensation code in the side entries/exits due to code motion [13, 14]. The schedulers described in this paper use the superblock [15] and hyperblock [16] as the global scheduling region. Wavefront scheduling uses a region consisting of a tree of basic blocks [17]. Meld scheduling extends the scope of the optimization without increasing the complexity associated with code motion by

Table 3. Ratio of scheduling steps and scheduling time (backtracking schedulers versus Cycle)

Proc. config	Scheduling steps (max)		Scheduling time (max)	
	ListBT	OperBT	ListBT	OperBT
111L1	1.00 (1.00)	1.00 (1.00)	0.86 (.92)	1.03 (1.04)
211L1	1.00 (1.00)	1.00 (1.00)	0.94 (.97)	1.02 (1.03)
111L2	1.05 (1.09)	1.56 (2.90)	0.88 (.93)	1.26 (1.71)
211L2	1.01 (1.03)	1.11 (1.42)	0.95 (.98)	1.04 (1.11)
111L3	1.07 (1.14)	1.43 (2.42)	0.89 (.94)	1.20 (1.56)
211L3	1.02 (1.02)	1.07 (1.23)	0.95 (.97)	1.04 (1.09)

allowing latency constraints to propagate outside the scheduling region [18].

Programs spend a significant fraction of the total execution time in loops and special scheduling techniques have been developed for loops [19, 20]. Rau developed the iterative modulo scheduler, which backtracks in a manner similar to the **ListBT** scheduler [21]. However the unique characteristics of modulo scheduling give rise to a very different set of algorithmic choices. Firstly, the iterative modulo scheduler may occasionally get locked in a repetitive orbit. The **OperBT** and **ListBT** schedulers are guaranteed to never revisit the same partial schedule and never get into a repetitive orbit. This guarantee is essential because, unlike the modulo scheduler, the option of increasing the initiation interval (II) and trying to schedule again is not available. Further, the primary optimization metric for the **ListBT** and **OperBT** schedulers is the schedule length, whereas schedule length is less important than II for modulo schedulers.

7 Conclusions

This paper motivates the need for backtracking schedulers by presenting processor features such as branch delay slots and resource conflicts that cannot be addressed adequately by non-backtracking schedulers. We present two backtracking schedulers that fill branch delay slots. The **OperBT** full backtracking scheduler picks operations in priority order and permits any operation to unschedule already scheduled operations. The **ListBT** selective backtracking scheduler enables backtracking only for certain operations. Experiments demonstrate that both backtracking schedulers considered successfully fill a significant fraction of branch delay slots, providing reduction in dynamic cyles of between 1-3%.

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] M.S. Schlansker and B.R. Rau, “EPIC: Explicitly Parallel Instruction Computing,” *IEEE Computer,* pp. 37-45, Feb. 2000.

[2] E. Fogel and J.C. Dehnert, “A code motion technique for scheduling bottleneck resources,” Proc. 1st Workshop on Media Processors and DSPs, pp. 13-21, 1999.

[3] V. Kathail, M.S. Schlansker, and B.R. Rau, “HPL PlayDoh Architecture Specification: Version 1.1,” Technical Report HPL-93-80 (R.1), Hewlett-Packard Laboratories, Feb. 2000 (originally published as "HPL PlayDoh Architecture Specification: Version 1.0", Feb. 1991).

[4] “Trimaran compiler infrastructure,” `http://www.trimaran.org` 1998.

[5] J.A. Fisher, “Global code generation for instruction-level parallelism: Trace scheduling-2,” Technical Report HPL-93-43, Hewlett Packard Laboratories, June 1993.

[6] S.G. Abraham, “Efficient backtracking instruction schedulers,” Technical Report HPL-2000-56, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techreports/2000/HPL-2000-56.html` May 2000.

[7] M. Rim and R. Jain, “Lower-bound performance estimation for the high-level synthesis scheduling problem,” *IEEE Trans. on CAD,* vol. 13, no. 4, 1994.

[8] M. Langevin and E. Cerny, “A recursive technique for computing lower-bound performance of schedules,” *ACM Trans. on Design Automation of Elec. Sys.,* vol. 1, no. 4, 1996.

[9] A.E. Eichenberger and W.M. Meleis, “Balance scheduling: Weighting branch tradeoffs in superblocks,” Proc. Int. Symp. Microarchitecture (MICRO-32), pp. 272-283, Haifa, Israel, Nov. 1999.

[10] P.B. Gibbons and S.S. Muchnick, “Efficient instruction scheduling for a pipelined architecture,” Proc. ACM SIGPLAN Symp. Compiler Construction, pp. 11-16, July 1986.

[11] H.S. Warren, “Instruction scheduling for the IBM RISC System 6000 processor,” *IBM J. Res. & Dev.,* vol. 34, no. 1, pp. 85-92, Jan. 1990.

[12] P.G. Lowney, et al., “The Multiflow trace scheduling compiler,” *J. Supercomputing,* vol. 7, no. 1, pp. 51-142, May 1993.

[13] D. Bernstein and M. Rodeh, “Global instruction scheduling for superscalar processors,” Proc. ACM SIGPLAN Symp. Prog. Lang. Des. & Impl. (PLDI), pp. 241-255, Toronto, Jun. 1991.

[14] S.M. Moon and K. Ebcioglu, “An efficient resource-constrained global scheduling method for superscalar and VLIW processors,” Proc. Int. Symp. Microarchitecture (MICRO-25), pp. 55-71, 1992.

[15] W.M. Hwu, et al., “The superblock: An effective technique for VLIW and superscalar compilation,” *J. Supercomputing,* vol. 7, no. 1, pp. 229-248, May 1993.

[16] S.A. Mahlke, et al., “Effective compiler support for predicated execution using the hyperblock,” *Proc. Int. Symp. Microarchitecture (MICRO-25),* pp. 45-54, 1992.

[17] J. Bharadwaj, K. Menezes, and C. McKinsey, “Wavefront scheduling: Path based data representation and scheduling of subgraphs,” Proc. Int. Symp. Microarchitecture (MICRO-32), pp. 262-271, Haifa, Israel, Nov. 1999.

[18] S.G. Abraham, V. Kathail, and B.L. Dietrich, “Meld scheduling: A technique for relaxing scheduling constraints,” *Int. J. Parallel Programming,* vol. 26, no. 4, pp. 349-381, 1998 (techical report version at `http://www.hpl.hp.com/techreports/97/HPL-97-39.html`).

[19] B.R. Rau, et al., “The Cydra 5 deparmental supercomputer: Design philosophies, decisions and trade-offs,” *IEEE Computer,* pp. 12-35, Jan 1989.

[20] J.C. Dehnert and R.A. Towle, “Compiling for the Cydra 5,” *J. Supercomputing,* vol. 7, no. 1, pp. 181-227, May 1993.

[21] B.R. Rau, “Iterative modulo scheduling,” *Int. J. Parallel Programming,* vol. 24, no. 1, pp. 3-64, Feb. 1996.