Efficient backtracking instruction schedulers

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

Abstract

Current schedulers for acyclic regions schedule operations in dependence order and never revisit or undo a scheduling decision on any operation. In contrast, backtracking schedulers may unschedule already scheduled operations, in order to make space for the operation currently being scheduled. Backtracking schedulers have the potential for generating better schedules, e.g. more effectively filling branch delay slots, but are more compile-time intensive and therefore, not considered practical for production use.

In this report, we first describe conventional cycle and list schedulers followed by two novel backtracking schedulers. The full-backtracking OperBT scheduler enables backtracking for all operations and un-schedules already scheduled operations to make space for the current operation, if, among other situations, there is no available slot that satisfies dependence constraints. This scheduler is effective in generating high quality schedules that for instance, successfully fill branch delay slots but likely backtracks too often. The selective backtracking ListBT scheduler enables backtracking only when scheduling certain types of operations, for which backtracking is likely to be advantageous, e.g. branches. When not scheduling these operations (or operations that were displaced through backtracking), the scheduler reverts to efficient scheduling in dependence order. The ListBT scheduler backtracks less often than the OperBT scheduler. Both schedulers successfully fill a large fraction of the branch delay slots and improve performance of the scheduled code significantly.

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

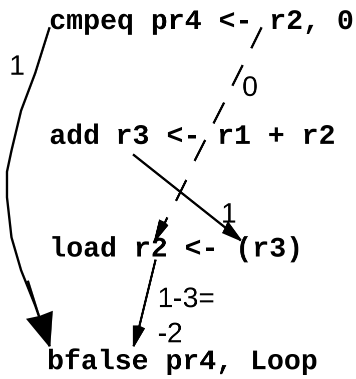
1 Introduction

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 many emerging mobile and communication applications where power consumption has to be optimized, EPIC processors are often preferred, partly because they do not require the additional power-consuming hardware to dynamically schedule instructions. Even though superscalar processors are less reliant on compiler technology, superscalar performance can also be improved through instruction scheduling. For the rest of this report, 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 schedules operations one by one, and does not undo or revisit scheduling decisions that were made previously. This report 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. Even a simple RISC processor may have five pipe stages for instruction fetch, align, decode/register read, execute, and write back respectively.




```
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>
```

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

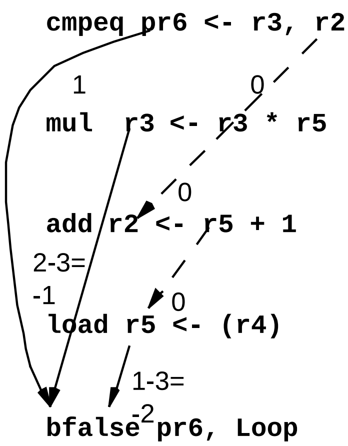
Figure 1: Conventional scheduler does not fill branch delay slots

Modern superscalar or VLIW processors may have as many as 15 or 20 stages. In these processors, branches are 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 bubble and a mis-prediction penalty of as much as 10 cycles. A supplement and/or alternative to branch prediction are 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 the latency of an edge from an arithmetic operation generating a live-out value to a branch. This edge latency must guarantee that the branch does not transfer control to another scheduling region before the arithmetic operation generates its live-out value. If the branch latency exceeds the arithmetic operation latency, this edge latency is negative, permitting the arithmetic operation to descend below the branch into its 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. In the example, 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. Solid arrows indicate true (flow) dependences and dashed arrows indicate anti-dependences. 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 (e.g. the `load` and `add`) are scheduled before the `bfalse`. 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. In this example, the `bfalse` has a higher priority than the `add` and displaces the scheduled `add`, which displaces the scheduled `load` in cycle 2, which in turn is eventually scheduled in 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%.

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 that are 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. But, 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. One alternative is to use a clustered approach with many register files with each cluster (group) of functional units using its own set of register files. But, this requires compiler techniques to map operations to clusters and may not be effective for some applications because of excessive inter-cluster communication. Another alternative is to move away from a dedicated write port for each functional unit to shared write ports. The compiler is now responsible for




```
0: cmpeq pr6 <- r3, r2
1: mul r3 <- r3 * r5
2: <empty - write port confl>
3: add r2 <- r5 + 1
4: load r5 <- (r4)
5: bfalse pr6, Loop
6: <empty delay slot>
7: <empty delay slot>
```

```
     Partial Schedule
0: cmpeq pr6 <- r3, r2
1: add r2 <- r5 + 1
2: mul r3 <- r3 * r5
3: load r5 <- (r4)

    Complete Schedule
0: cmpeq pr6 <- r3, r2
1: add r2 <- r5 + 1
2: bfalse pr6, Loop
3: mul r3 <- r3 * r5
4: load r5 <- (r4)
```

Figure 2: Backtracking schedulers can handle resource conflicts

scheduling operations so that there is no resource conflict on the write ports, even among operations with disparate latencies.

The example in Figure 2 demonstrates why conventional scheduling approaches may not handle write port conflicts effectively. In this example, we assume a single-issue processor where the `multiply` and `add` share the same write port and their latencies are two and one respectively. The `cmpeq` operation has the highest priority and is scheduled in cycle 0. A conventional scheduler then schedules the `multiply` and finds that it cannot schedule the `add` in the next cycle due to the resource conflict on the write port. None of the other operations can be issued in cycle 1. Thus, the schedule length is 8. A backtracking scheduler also schedules the `multiply` in cycle 1. Since the delay in scheduling the `add` in cycle 3 extends the entire schedule, the `add` displaces the `multiply`. The `load` uses distinct ports and can be issued in cycle 3. Finally, as in the example in Figure 1, the branch displaces other scheduled operations and is scheduled eventually in cycle 3. The schedule length is reduced by three cycles, two due to filling branch delay slots and one due to filling a write port conflict slot.

Though backtracking schedulers can be more effective than conventional schedulers for a variety of reasons, this report focuses on how backtracking schedulers can fill branch delay slots. An alternative to backtracking schedulers is to use peephole optimization strategies that work locally across a few instructions. For instance, a post-scheduling peephole strategy to fill delay slots is to go through the schedule and first identify branches whose delay slots are unfilled.

Then, we attempt to swap such branches with other regular operations scheduled in earlier cycles without violating dependence and resource constraints.

The first problem with such peephole optimization strategies is that they 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, where each design is not a big revenue generator, it may not be cost-effective to develop these kinds of compiler optimizations. For instance, the objective of the PICO project is to automatically develop a customized processor for an application The PICO system walks over a large design space[2, 3] generating a machine-description driven compiler for each design point. Any compiler optimization must be automatically generated for each design point. It is difficult to automatically develop and tune a peephole optimization based on a machine description. The second problem is the local scale of peephole optimizations. Consider superblock scheduling where the scheduling region is a chain of basic blocks with one entry point. Moving the branch up in the first basic block of the superblock may relax the dependences between the branch and subsequent operations further down in other basic blocks. It may be possible to move these operations up in the schedule too. But such large-scale scheduling changes are difficult to encode as peephole optimizations.

A basic block scheduler may first schedule the single branch exit and schedule all other operation in reverse order relative to the branch. This approach is successful in filling branch delay slots when there is a single branch exit. Due to the limited amount of instruction-level parallelism in a single block, compilers tend to use larger scheduling regions such as superblocks, which have several branch exits. Though basic block schedulers are a core component of global scheduling approaches, the delay slot filling approach does not extend to global schedulers.

In summary, deeper pipelines and the power requirements of prediction hardware motivate exposing all or some of the branch latency to the compiler. Current 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 [4, 5]. However, backtracking schedulers for basic block and superblock schedulers have not been developed and their efficacy and compile-time complexity have not been studied previously.

In this report, we first describe the overall scheduling model consisting of a processor model, scheduler input and output model and the overall objective. Then, we describe the operation of a conventional **Cycle** scheduler that schedules operations cycle by cycle or in VLIW parlance, instruction by instruction. The **List** scheduler schedules ready operations in priority order, not necessarily cycle by cycle. We demonstrate that these conventional schedulers, viz. **Cycle** scheduler and **List** 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 unschedules operations unnecessarily. 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 put all four schedulers into a single software framework, which is amenable to experimentation.

2 Scheduling Model

2.1 Processor architecture

We use a family of VLIW processors based on the HPL-PD architecture [6]. 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. Thus, an instruction for a 312 processor may contain up to three integer operations and up to a total of six operations. We assume that functional units are fully pipelined. Thus, operations from different instructions (necessarily issued in distinct cycles) do not compete for resources.

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

Additionally, a processor can issue a branch operation on 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 report.

2.2 Scheduler input

We first use the Impact compiler from the University of Illinois that is part of the Trimaran compiler infrastructure [7] 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 IMPACT 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, which is also part of the Trimaran compiler infrastructure [7], takes the input in superblock form, performs data-flow analyses and constructs dependence graphs. Live-in values at the superblock entry are associated with data merge (DM) operations. Live-out values at each superblock exit are associated with data switch (DS) operations. The first operation of each constituent basic block is a control-merge (CM) operation. Control-merge, data-merge and data-switch operations are referred to as pseudo-operations because they do not map to actual assembly-level operations. The regular operations that ultimately map to assembly-level operations are called real operations. The nodes in the dependence graph are composed of all real as well as all pseudo operations. Data-merge operations are associated with and assume the schedule times of their corresponding control-merge operation. Similarly, data switch operations are associated with and assume the schedule times of their corresponding branch operation.

An edge between two operations is annotated with a latency indicating the minimum separation in their schedule times and hence 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. Figure 3 shows a dependence graph and a valid schedule on a 111L3 processor.

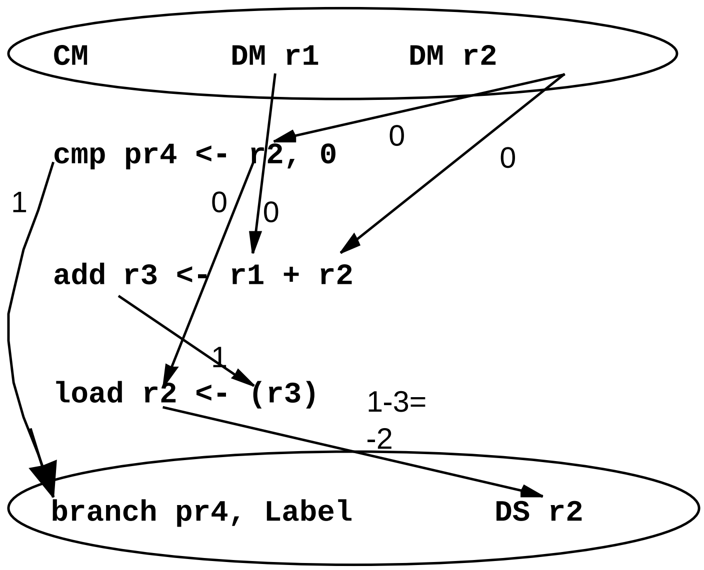
2.3 Scheduler output

The scheduler assigns a valid issue cycle for each operation in the superblock. The scheduler assigns resources to each operation so that there are no resource conflicts between scheduled operations. The scheduler encodes the resource assignment for each operation by associating it with a machine-specific opcode. In addition, the scheduler orders the operations within each cycle so that all zero-cycle dependencies flow from left to right. Finally, the scheduler threads all the non data-switch/merge operations in the superblock into a single chain in which operations are sorted in increasing order of schedule cycles and further all operations scheduled in the same cycle are contiguous and in the prescribed left to right order.

2.4 Scheduler objectives

The schedule generated by the scheduler must satisfy the following constraints:

1. dependence edge constraints are satisfied, i.e. for each dependence edge the issue cycle of the destination operation minus the issue cycle of the source operation is not less than the edge latency




```
0: CM; cmp pr4 <- r2, 0
1: branch pr4, Label
2: add r3 <- r1 + r2
3: load r2 <- (r3)
```

Figure 3 Internal representation of dependence graph

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 operations of that type. For instance in a 312L3 processor, no more than two memory operations are scheduled in any cycle. Also, at most a single branch is scheduled in a cycle and the number of integer operations plus branches does not exceed three.

The scheduler optimizes the profile-weighted execution time of each superblock. The execution time of a particular superblock 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 times its exit time. The exit time of a branch is the sum of the corresponding branch’s issue cycle and the branch latency.

The performance measurements 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 mis-predictions, TLB misses, etc. These other factors are expected to be similar with or without backtracking schedulers and are not expected to affect significantly the accuracy of our evaluation of backtracking schedulers vis-a-vis conventional cycle scheduling. 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 report 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 will affect the performance of all schedulers in a similar manner.

3 Conventional schedulers

In this section, we describe common pre-scheduling steps as well as two conventional schedulers that do not backtrack. The **Cycle** scheduler schedules all the operations to be issued in a particular cycle before going on to the next cycle. The **List** scheduler schedules operations in dependence order, ensuring that all predecessors of an operation have been scheduled before an operation is considered for scheduling. Secondarily, the **List** scheduler schedules operations in static priority order.

3.1 Common pre-scheduling steps

The **Cycle** scheduler as well as the other schedulers described in this report starts 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 dependence graph of a superblock is acyclic. We determine the early times of all operations by visiting each operation exactly once in topological order, where we visit the ancestors of an operation before visiting an operation. When we visit an operation, we already have computed the early times of its predecessors and the early time of the operation is the maximum over all incoming edges of the sum of the predecessor's early time and the edge latency.

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. We determine the late times of all operations by starting from the exit

E and visiting the operations in the dependence graph of a superblock in reverse topological order, where we visit the descendants of an operation before visiting an operation. The late times of operations that are not visited in this traversal are set to *maxheight*, where *maxheight* is the maximum early time among all operations in the superblock.

The late time of an operation with respect to an exit E represents how low or late an operation can be issued while still being able to schedule E at its early time on an infinite resource processor. The *height* of an operation with respect to an exit E is *maxheight* minus the late time of that operation with respect to E. The *weighted height* of an operation is the sum over all superblock exits, E, of the product of the profiled weight of E times 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 [8].

3.2 Conventional Cycle Scheduler

Before we describe the main scheduling loop of the **Cycle** scheduler, we describe some concepts and data structures.

3.2.1 Concepts and main scheduling loop

The *CurrentCycle* is the cycle in which operations are being scheduled currently by the **Cycle** scheduler. The *CurrentCycle* is initially set to 0 and incremented when no more operations can be scheduled in the *CurrentCycle* 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 an 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 an operation in a particular cycle.

A *ready operation* is an operation whose predecessors have been scheduled and a *ReadyList* is a 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 constraint on its incoming edges will not be violated by scheduling it in the *CurrentCycle*, i.e. the difference between the

CurrentCycle and its predecessor operations' *ScheduleCycle* is not less than the edge latency. 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 of the main scheduling loop, we recompute the *CCReadyList*, the list of all ready operations for the current cycle. We discuss below how to incrementally recompute *ReadyList* and *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 of the main scheduling loop. If the *CCReadyList* has one or more operations, we remove the highest priority operation from the *CCReadyList*. If this operation has no resource conflicts with already scheduled operations, we schedule the operation in the *CurrentCycle*. Otherwise, this operation 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.

3.2.2 Incremental recomputation of *ReadyList* and *CCReadyList*

The initial and any non-incremental computation of the *ReadyList* requires visiting all the operations in the scheduling region and retaining those operations that are not themselves scheduled but whose predecessors are all scheduled. The *CCReadyList* computation requires iterating through the *ReadyList* and selecting those operations whose *EarlyCycle* is not less than the *CurrentCycle* and whose predecessors are scheduled sufficiently in advance that all dependence edge constraints will be satisfied if the operation is scheduled in the *CurrentCycle*.

We now discuss how we can incrementally recompute *ReadyList* and *CCReadyList*. The above algorithm requires computing the *CCReadyList* after each scheduling iteration, which:

1. finds that *CCReadyList* is empty and increments *CurrentCycle*,
2. removes an operation from *CCReadyList* and schedules it for *CurrentCycle*,
3. removes an operation from *CCReadyList* and increments its *EarlyCycle*.

In case 1, the *ReadyList* is up to date, but since the *CurrentCycle* is incremented, we recompute *CCReadyList* by iterating through all the operations in *ReadyList*. In case 2, the scheduling of an operation may create other ready operations and *CurrentCycle* ready operations. We find all these operations by iterating through the successors of the scheduled operation. In case 3, no additional ready operations are created, so the *ReadyList* and *CCReadyList* are up to date. Note that case 2 above is the most time-consuming of the three cases. We have to examine all the successors of the scheduled operation and further iterate over all the predecessors of this successor to determine if the successor has become ready.

By maintaining a count of unscheduled incoming edges (*NumUnsched*) for each operation, we obtain a further improvement in incremental recomputation of *ReadyList* and *CCReadyList* in case 2. An *unscheduled incoming edge* is an incoming dependence edge from an operation that is currently not scheduled. Before entering the main scheduling loop, we initialize each operation's *NumUnsched* with the total number of incoming edges from predecessors. When an operation is scheduled, we visit each of its successors and decrement their *NumUnsched* and update their *EarlyCycle*. If *NumUnsched* is 0, then this successor operation is now a ready operation and is moved to the *ReadyList* and possibly to the *CCReadyList*. When we maintain *NumSched*, we do not need to iterate through the predecessors of a successor operation.

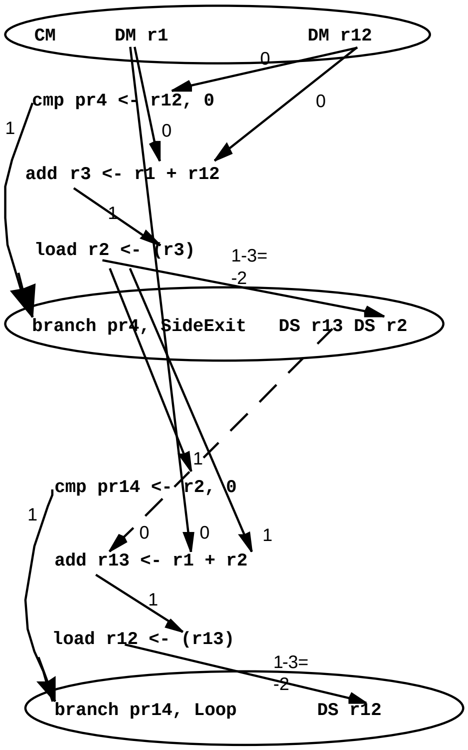
Consider the example in Figure 1, where the **Cycle** scheduler generates the inefficient schedule of length six. Initially, the only ready operation, CM, for cycle 0 is scheduled in cycle 0. This operation does not use issue resources and makes the `cmpeq` and `add` ready for cycle 0. The priority of `cmpeq` is higher because its height is one above the branch versus –1 for the `add`. Therefore, the `cmpeq` is first scheduled in cycle 0 followed by the `add` in cycle 1. Scheduling the `add` in cycle 1, makes the `load` ready for cycle 2. Scheduling the `load` in cycle 2, makes the `bfalse` ready for cycle 1. Because *CurrentCycle* is now at cycle 2, the `bfalse` is only considered for scheduling in cycle 2. But, in a single-issue processor, the `bfalse` resource conflicts with the load. So, its *EarlyCycle* is incremented to 3 and the `bfalse` is scheduled in cycle 3.

3.2.3 Cycle scheduling and peep-hole optimizations

An alternative is to use a post-scheduling peephole optimization strategy to improve the quality of the schedule while retaining the simplicity of the cycle scheduler. A *peephole optimization* recognizes common inefficient patterns of operations and replaces them with a more efficient pattern. In the context of branch scheduling, one can recognize branches whose delay slots are unfilled and consider the validity of a swap with another operation within a neighborhood of the unfilled delay slots of the branch. In the example in Figure 1, the branch has two unfilled delay slots. We first consider swapping the schedules for the `bfalse` and the `add`, but this violates the dependence edge from the `add` to the `load`. We then consider swapping the schedules for the `branch` and the `load`, which is valid. The resulting peephole optimized schedule fills one delay slot. Of course, a more comprehensive peephole optimization strategy can consider three-way swaps of operations and perhaps fill both delay slots.

Figure 4(a) shows the dependence graph for a superblock consisting of two unrolled blocks from the example in Figure 1. Figure 4(b) show the 12 cycle long schedule generated by the **Cycle** scheduler. A peephole optimizer that does a three-way swap may reorganize the schedule and shorten it to 10 cycles as shown in Figure 4(c). But, there are still two empty delay slots, which can be filled only by moving up all the operations in cycles 6 through 9 as in Figure 4(d). In this case, all the operations in cycles 6 through 9 move up as a group, but in general that may not be the case. Therefore, we may not be able to generate the optimum schedule of 8 cycles shown in Figure 4(d).

Though a peephole optimization strategy may be effective in filling some of the delay slots, it may not produce the same quality of schedules that a more sophisticated backtracking scheduler can generate and it may require expensive manual fine-tuning for each processor model. One drawback of peephole optimization techniques is that they have to be specifically developed and tuned for a particular machine with its set of resources and latencies. For instance, one may develop a simple and effective two-way swap for a processor model with a branch latency of two. But one may be forced to develop a more complex three-way swap when the branch latency changes to three. Another drawback of peephole optimizers is that they do not work well when the scope of the optimization is enlarged.



(a)

```
0: CM; cmp pr4 <- r12, 0
1: add r3 <- r1 + r12
2: load r2 <- (r3)
3: branch pr4, SideExit
4: <empty delay slot>
5: <empty delay slot>
6: cmp pr14 <- r2, 0
7: add r13 <- r1 + r2
8: load r12 <- (r13)
9: branch pr14, Loop
10: <empty delay slot>
11: <empty delay slot>
```

(b)

```
0: CM; cmp pr4 <- r12, 0
1: branch pr4, SideExit
2: add r3 <- r1 + r12
3: load r2 <- (r3)
4: <empty delay slot>
5: <empty delay slot>
6: cmp pr14 <- r2, 0
7: branch pr14, Loop
8: add r13 <- r1 + r2
9: load r12 <- (r13)
```

(c)

```
0: CM; cmp pr4 <- r12, 0
1: branch pr4, SideExit
2: add r3 <- r1 + r12
3: load r2 <- (r3)
4: cmp pr14 <- r2, 0
5: branch pr14, Loop
6: add r13 <- r1 + r2
7: load r12 <- (r13)
```

(d)

Figure 4: Peephole optimizations

3.3 List scheduler

Like the **Cycle** scheduler, the **List** scheduler also maintains *ReadyList*, a list of operations whose predecessors have already been scheduled. In each iteration of the main scheduling loop, it selects the highest static priority operation from the *ReadyList* and schedules it in the earliest cycle possible that satisfies all resource constraints, starting with the *EarlyCycle* of the operation. In the **List** scheduler, *EarlyCycle* is the earliest cycle that an operation can be scheduled without violating any dependence constraints with its scheduled predecessors.

Once an operation is scheduled, the *ReadyList* is updated. As described in Section 3.2.2, each operation has an associated *NumUnsched* count, indicating the number of incoming dependence edges whose source operations have not been scheduled. After an operation is scheduled, we iterate through all its outgoing edges and decrement the *NumUnsched* count and update the

EarlyCycle of the successor operations. We move any operation whose *NumUnsched* count reaches zero to the *ReadyList*.

In the **List** scheduler, operations are not necessarily scheduled cycle by cycle. We may schedule an operation in cycle 4 and then schedule the next operation in cycle 2. However, since we schedule an operation's predecessors prior to scheduling an operation, we are guaranteed that we can always make forward progress and backtracking is not needed. Further, **List** scheduling is very efficient in compile time because once an operation is selected for scheduling it is always scheduled.

For the example in Figure 1, the **List** scheduler generates the same inefficient schedule of length six as the **Cycle** scheduler. As in the **Cycle** scheduler, the `cmpeq`, `add` and `load` are scheduled in cycles 0, 1, 2 respectively. Scheduling the `load` in cycle 2, makes the `bfalse` ready for cycle 1. Therefore, unlike the **Cycle** scheduler, the **List** scheduler attempts to schedule the `bfalse` starting at cycle 1. We fail to schedule the `bfalse` in cycles 1 and 2 due to resource conflicts with the already scheduled `add` and `load`. The `bfalse` is eventually scheduled in cycle 3. If there were no resource conflicts in the earlier cycles, the **List** scheduler would have scheduled the `bfalse` earlier, filling some of the delay slots. But, in this example, the **List** scheduler does not fill branch delay slots.

4 Backtracking schedulers

In this section, we describe two novel backtracking schedulers; **OperBT** and **ListBT**. The full-backtracking **OperBT** scheduler enables backtracking for all operations and unschedules already scheduled operations to make space for the current operation. The selective backtracking scheduler **ListBT** enables backtracking only when scheduling certain types of operations, for which backtracking is likely to be advantageous, e.g. branches. When not scheduling these operations (or operations that were displaced through backtracking), the scheduler reverts to efficient scheduling in dependence order like the **List** scheduler

4.1 Common concepts

As in the conventional scheduler, we compute early times, late times and priorities for each operation before entering the main scheduling loop. Unlike the conventional scheduler, not all predecessors of an operation may be scheduled at the time when an operation is considered for scheduling. Therefore, the *EarlyCycle* of an operation is the maximum of its early time and the earliest time that an 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 that an operation can be issued while satisfying all the dependence edges to successor scheduled operations.

A *resource conflict* prevents the scheduling of *CurrentOperation* in *CurrentCycle* if the scheduled operations have reserved resources that are required by the *CurrentOperation*. The *CurrentOperation* has a resource conflict with a particular scheduled operation if that operation has reserved some resources that are required by *CurrentOperation* and there are no other free resources of that type available. A *dependence conflict* indicates that a dependence edge latency from a scheduled operation to the *CurrentOperation* is not satisfied if the *CurrentOperation* is scheduled at *CurrentCycle*. The *set of conflicting operations* is the set of all operations that either have resource or dependence conflicts with *CurrentOperation*.

Since the 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. Even when a range of cycles is available, the operation may have resource conflicts that prevent it from being scheduled in these cycles.

In such situations, we need a mechanism to make forward progress. The backtracking schedulers may unschedule other conflicting operations in order to schedule *CurrentOperation* in a particular cycle. A scheduled operation is *unscheduled* by removing its association with a particular issue cycle, releasing resources that 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 the operation was last scheduled. While iterating through the cycles ranging from *EarlyCycle* through *LateCycle*, the backtracking scheduler may *displace* schedule by first unscheduling lower priority conflicting operations and then scheduling *CurrentOperation*. If the scheduler is unable to normal or displace schedule *CurrentOperation* in the cycles ranging from *EarlyCycle* through *LateCycle*, the backtracking scheduler *forcibly* schedules by first removing operations that conflict with *CurrentOperation* at a chosen *ForceCycle* and then scheduling *CurrentOperation* in *ForceCycle*. The forcible scheduling mechanism ensures that once we select a *CurrentOperation*, we are always able to successfully schedule *CurrentOperation*, even if that requires unscheduling other operations.

The next problem that we may encounter is that the scheduler gets into 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. In order to avoid such termination problems, we maintain *AttemptedCycle* with each operation. *AttemptedCycle* is the last attempted cycle that we forcibly scheduled that operation. When we first unschedule a particular operation, we set its *AttemptedCycle* to *ScheduleCycle*`-1`, where *ScheduleCycle* is the cycle in which the operation was scheduled. *ForceCycle* is the cycle in which we forcibly schedule *CurrentOperation* and we choose *ForceCycle* to be the maximum of *EarlyCycle* and *AttemptedCycle*`+1` and set *AttemptedCycle* to the updated *ForceCycle*. Thus, the *ForceCycle* in which a particular operation is forcibly scheduled is guaranteed to increase monotonically.

4.2 OperBT scheduler

The **OperBT** scheduler maintains an *UnschedList*, a sorted list of operations that have not been scheduled in priority order, where the priority is as computed for instance by the weighted priority algorithm. The main scheduling loop iterates until *UnschedList* is empty. In each iteration, we remove the highest priority operation from the *UnschedList* and set it to *CurrentOperation*. We compute the *EarlyCycle*, *LateCycle* and *ForceCycle* of *CurrentOperation*. We iterate through the cycles from *EarlyCycle* through *LateCycle* attempting to schedule an operation. We normal schedule the operation at *CurrentCycle*, if resources are available. Otherwise, we displace schedule *CurrentOperation* at *CurrentCycle* if *CurrentCycle* is not less than *ForceCycle* and 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 operation after iterating through the cycles ranging from *EarlyCycle* through *LateCycle*, we force schedule *CurrentOperation* at *ForceCycle*. Regardless of the relative

```
Initialize EarlyCycle, LateCycle and compute priorities of operations
while (CurrentOperation = UnschedList.pop())
  Compute EarlyCycle and LateCycle for CurrentOperation
  ForceCycle = max (AttemptedCycle+1, EarlyCycle)
  success = FALSE
  for (CurrentCycle ranging from EarlyCycle through LateCycle)
    if (resources required by CurrentOperation available)
        Schedule CurrentOperation in CurrentCycle
        success = TRUE
        break
    elsif ( ( CurrentCycle >= ForceCycle) AND
            (HasHigherPriority (CurrentOperation, CurrentCycle)))
        Unschedule conflicting operations and push them back into UnschedList
        Schedule CurrentOperation in CurrentCycle
        success = TRUE
        break
    endif
  endfor
  if (success = FALSE)
    Unschedule conflicting operations at ForceCycle
         and push them back into UnschedList
    Schedule CurrentOperation in ForceCycle
    Set AttemptedCycle to ForceCycle for CurrentOperation
  endif
endwhile
```

Figure 5: OperBT scheduler

priority of the resource and dependence conflicting operations, we unschedule all the conflicting operations and forcibly schedule the *CurrentOperation* in the *ForceCycle*. The pseudo-code for the **OperBT** scheduling algorithm is shown in Figure 5.

Lemma 1

The OperBT scheduler does not deadlock and does terminate.

Proof:

Once we select a *CurrentOperation* from *UnschedList*, we always schedule it, either in the cycles ranging from *EarlyCycle* through *LateCycle* or at *ForceCycle*. Therefore, the scheduler does not deadlock.

For our purposes, operation latency is the maximum number of cycles for which an operation uses resources in the processor. Since register writes (or forwarding) also use resources, the operation latency also limits the latency of outgoing edges. Let *MaxLatency* be the maximum of the operation latency of any operation and the edge latency between any two operations. Let *SL* be the sum of the absolute values of the dependence edge latencies and the operation latencies. Let *SLB* be *SL* times the number of operations in the dependence graph.

In each scheduling step, the scheduler may (1) schedule an operation without unscheduling (*normal*), (2) unschedule lower priority operations and schedule an operation at *CurrentCycle* (*displace*), (3) unschedule high priority operations and schedule an operation at *ForceCycle* (*force*). Normal scheduling reduces the size of *UnschedList* that contains operations that are not currently scheduled. Displace and force scheduling may increase the size of *UnschedList*. Let *PR* be the priority of the highest priority operation in *UnschedList*.

Assume that the scheduler does not terminate. The scheduler must intersperse force scheduling that may increase *PR* with normal and displace scheduling that never increases *PR*. Otherwise, *PR* decreases indefinitely and the *UnschedList* becomes empty terminating the scheduler.

Therefore it must forcibly schedule at least one operation *A* more than *SLB* times. Since the *AttemptedCycle* is at least advanced by `1` on each forcible schedule, the *ScheduleCycle* of *A* must advance past *SLB*.

Consider a partial schedule, *PS*, in which *A* has a *ScheduleCycle* more than *SLB*. The slack of a dependence edge is defined when both its source and destination operations are scheduled in the partial schedule and is equal to the difference between the destination operation's *ScheduleCycle* and the sum of the source operation's *ScheduleCycle* and the edge latency. Obviously, the slack of some dependence edges in *PS* in a path from the start operation to *A* have a slack more than *SL*.

Consider the set *C* of scheduled operations in *PS* that are connected to the Start operation by dependence edges with slack less than *SL*. This set does not contain *A* because *A*'s *ScheduleCycle* is more than *SLB* and at least one edge along a path from the Start operation to *A* must have a slack more than *SL*. Consider the non-empty set *D* of the remaining scheduled operations in *PS*, which includes *A*.

Consider the subgraph of the dependence graph induced by the operations in *D*. This subgraph is acyclic because the entire dependence graph is acyclic. Choose a root, *R*, of this subgraph, *SG* (an operation with no incoming edges from operations in *D*). When *R* was last scheduled, it must have had a scheduled predecessor *P* scheduled within *MaxLatency* of *R*'s current *ScheduleCycle* (if *R* did not have resource conflicts with other operations) or within *SL* of *R*'s current *ScheduleCycle* (if *R* had resource conflicts with potentially all the operations). Otherwise, the *EarlyCycle* of *R* would at most be *MaxLatency* more than the maximum *ScheduleCycle* of the operations in *C* and the operation *R* would have been scheduled with less than a slack of *SL* on its incoming edges.

This scheduled predecessor *P* is currently unscheduled because otherwise *R* would not be a root of the subgraph *SG*. We now consider the partial schedule *PS'*, in which *P* was last scheduled. We recursively apply the argument we used for *R* in *PS* to *P* in *PS'*. Since the number of such currently unscheduled predecessors is finite, we conclude that some ancestor of R currently unscheduled in PS, did not have any scheduled predecessors when it was last scheduled within *SL* of its last *ScheduleCycle* and yet it was scheduled with a slack of more than *SL* with respect to the operations in *C*, something the **OperBT** scheduler would not do. This leads to a contradiction and the **OperBT** scheduler does terminate. ■

Experimental results indicate that the **OperBT** scheduler is very effective in filling the delay slots of branches. Further, the **OperBT** scheduler is guaranteed to terminate with a complete schedule. 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. Among the ready operations, the **ListBT** scheduler selects operations based on priority, as computed prior to entering the main scheduling loop. Finally, the **ListBT** scheduler backtracks by enabling certain operations to unschedule lower priority scheduled operations.

Prior to entering the main scheduling loop, the **ListBT** scheduler calculates early and late times and priorities for each operation. Additionally, only certain operations are permitted initially to unschedule other operations and the *AttemptedCycle* of these operations are set to early time minus unity. The *ReadyList*, the set of operations whose predecessors are scheduled, is initially set to the start operation of the superblock.

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 operations.

Initially, operations are scheduled in dependence order. Therefore the *LateCycle* of an operation is infinity and the operation can always be successfully scheduled in some cycle. Once an operation is unscheduled, it may have a finite range of valid cycles between its *EarlyCycle* and *LateCycle*. If unscheduling is disabled for this operation, it may not be possible to successfully schedule this operation, leading to a deadlock. Therefore, unscheduling is enabled for any operation that is unscheduled for the first time and its *AttemptedCycle* is set to the *ScheduleCycle*`-1`. The *ForceCycle* of an operation is the minimum cycle in which an operation may be forcibly scheduled by unscheduling other operations. The *ForceCycle* of an operation is the maximum of its *EarlyCycle* and *AttemptedCycle*`+1`.

The main scheduling loop iterates until the *ReadyList* is empty. In each iteration, we remove the highest priority operation from the *ReadyList* and set it to be the *CurrentOperation*. We find *EarlyCycle* and *LateCycle* for this operation; *LateCycle* is infinity unless this operation has been unscheduled. We iterate through the range from *EarlyCycle* and *LateCycle*. If resources are available for the *CurrentCycle*, we schedule the operation for the *CurrentCycle*. Otherwise, if unscheduling is enabled for *CurrentOperation* and the *CurrentCycle* is not less than the *ForceCycle* of the operation, and the conflicting operations are lower in priority than *CurrentOperation*, we first unschedule all conflicting operations and then forcibly schedule the operation for the *CurrentCycle*. If we do not schedule the operation in the range *EarlyCycle* through *LateCycle* then the operation must be an unscheduled operation (because only such operations have *LateCycle* less than infinity) and unscheduling must be enabled for this operation. We forcibly schedule the operation at *ForceCycle*, unscheduling operations that have dependence or resource conflicts with *CurrentOperaton* being scheduled at *ForceCycle*. When an operation is forcibly scheduled, its *AttemptedCycle* is set to the *ForceCycle*. The pseudo-code for the ListBT scheduling algorithm is shown in Figure 6.

As in **OperBT**, the previous backtracking scheduler, 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.

5 Framework for scheduling algorithms

Though we have discussed four schedulers that differ substantially in their behavior, they require a fair amount of shared functionality. This common part is a substantial fraction of the software code base for the scheduler. There are the common pre-scheduling steps such as early and late time calculation and priority calculation. There are also some common utilities that are required in the main scheduling loop such as calculating the *EarlyCycle* and *LateCycle*, finding the conflicting operations, determining if an operation has higher priority than its conflicting operations, forcibly scheduling an operation in a cycle, scheduling and unscheduling operations. This common part also includes various post-scheduling steps such as reordering/rethreading the operations in schedule order, marking operations that have speculated, treating branches for different machine models and compiling models and checking the schedule for correctness.

This section puts all the schedulers that we discussed in the same software framework. This framework helps us to understand the different components of a scheduling algorithm and also enables us to factor the code base effectively, so that the common parts of the scheduler are leveraged across all schedulers. In our framework, there are three distinguishing characteristics

```
Initialize EarlyCycle, LateCycle and compute priorities of operations
ReadyList = Start operation
while (CurrentOperation = ReadyList.pop())
  Compute EarlyCycle and LateCycle for CurrentOperation
  ForceCycle = max (AttemptedCycle+1, EarlyCycle)
  success = FALSE
  for (CurrentCycle ranging from EarlyCycle through LateCycle)
    if (resources required by CurrentOperation available)
        Schedule CurrentOperation in CurrentCycle
        Update ReadyList with ready successors of CurrentOperation
        success = TRUE;
        break
    elsif ((unscheduling enabled for CurrentOperation) AND
          (CurrentCycle >= ForceCycle) AND
          (HasHigherPriority (CurrentOperation, CurrentCycle)))
        Unschedule conflicting operations and update ReadyList
        Enable unscheduling for conflicting operations
        Schedule CurrentOperation in CurrentCycle and update ReadyList
        success = TRUE
        break
    endif
  endfor
  if (success = FALSE)
    Unschedule conflicting operations at ForceCycle and update ReadyList
    Enable unscheduling for conflicting operations
    Schedule CurrentOperation in ForceCycle and update ReadyList
    Set AttemptedCycle to ForceCycle for CurrentOperation
  endif
endwhile
```

Figure 6: ListBT scheduler

about a particular scheduler, viz. which operation is chosen next for scheduling, which cycles do we attempt to normally schedule an operation, and which cycle, if any, can we forcibly schedule an operation by unscheduling other operations.

The `Priority` class is responsible for choosing the next operation for scheduling. Before entering the main scheduling loop, the `Priority` class is initialized with a new scheduling region. In the main scheduling loop, the `Priority.pop()` function delivers the next operation to be scheduled, accounting for dependence, priority and other constraints. If an operation is unscheduled or is not successfully scheduled, the `Priority.push()` function puts the operation back on the unscheduled list of operations.

There are several derived classes (specializations) of the `Priority` class that are used with the specific schedulers discussed earlier. The `PriorityCycle` class maintains the *ReadyList*, a list of ready operations and *MinCycle*, the minimum cycle that any of these ready operations can be scheduled without violating dependence constraints with predecessor scheduled operations. The `PriorityCycle` also maintains *CCReadyList*, the list of ready operations that can be issued at *MinCycle*. The `PriorityCycle.pop()` function delivers the ready operation with the highest priority among the operations in *CCReadyList*. The `PriorityCycle.push()` function increments the *EarlyCycle* of an operation and puts the operation back on the *ReadyList*.

The `PriorityDependence.pop()` delivers highest priority operation among all ready operations (not necessarily those schedulable at *MinCycle*). The `PriorityDependence.push()` updates the *AttemptedCycle* of an operation to its current *ScheduleCycle*`-1`, if it is the first time that the operation is being unscheduled and puts the operation back on the unscheduled set of operations.

The `PriorityStatic.pop()` function delivers the highest priority operation among all operations (not necessarily among the ready operations). The `PriorityStatic.push()` puts the operation back on the unscheduled list of operations with an *AttemptedCycle* of *ScheduleCycle*`-1`, if it has not been scheduled previously.

The `Cycle` class is responsible for deciding the *CurrentCycle* in which *CurrentOperation* should be scheduled. At the beginning of each iteration of the scheduling loop, the `Cycle` class is initialized with the *CurrentOperation* and its *EarlyCycle* and *LateCycle*. The main scheduling loop then repeatedly invokes `Cycle.nextCycle()` to obtain `CurrentCycle` till this function returns `NULL`. The `CycleEtime.nextCycle()` returns *EarlyCycle* on the first invocation and `NULL` thereafter. The `CycleSequential.nextCycle()` returns the value of an internal counter that is initially set to `EarlyCycle` and that increments following each invocation. After the counter advances past *LateCycle*, `CycleSequential.nextCycle()` returns `NULL`.

The `Unschedule` class is responsible for deciding if the scheduler should forcibly schedule *CurrentOperation.* At the beginning of each iteration of the scheduling loop, the `Unschedule` class is initialized with *CurrentOperation*, *EarlyCycle* and *LateCycle*. The `Unschedule.getForceCycle()` either returns a *ForceCycle* or returns `NULL` to indicate that the scheduler should not forcibly schedule the *CurrentOperation*. The `UnscheduleNever.getForceCycle()` always returns `NULL` and never permits forcible scheduling. The `UnscheduleSometimes.getForceCycle()` returns `NULL` if the *CurrentOperation* has an un-initialized value for *AttemptedCycle*, otherwise it returns the maximum of *EarlyCycle* and (*AttemptedCycle*`+1`). The `UnscheduleAlways.-getForceCycle()` returns the maximum of *EarlyCycle* and `(`*AttemptedCycle*`+1)` always.

The `Scheduler` class constructs specialized versions of the `Priority`, `Cycle` and `Unschedule` classes. Prior to scheduling each region, the `Scheduler` initializes the `Priority` class. In the main scheduling loop, the `Scheduler` repeatedly pops an operation from the `Priority` class and sets it to the *CurrentOperation*. When there are no more operations to be scheduled, the `Priority` class returns `NULL` and the scheduling loop terminates. The Scheduler calculates the *EarlyCycle* and *LateCycle* of *CurrentOperation* based on scheduled predecessors and successors of *CurrentOperation* respectively. The Scheduler initializes the `Cycle` and `Unschedule` classes. Then, it relies on the `Cycle` class to iterate over all the valid cycles for *CurrentOperation*, until *CurrentOperation* is successfully scheduled or all the valid cycles are exhausted. In each valid *CurrentCycle*, the `Scheduler` first checks to see if sufficient resources are available to schedule *CurrentOperation*. If so, it schedules *CurrentOperation* in *CurrentCycle*. Otherwise, it next checks to see if `Unschedule.getForceCycle()` returns a non-`NULL` *ForceCycle* and if this *ForceCycle* is not more than *CurrentCycle*, and if the conflicting operations (*UnscheduleSet*) with *CurrentOperation* are of lower priority relative to *CurrentOperation*. If so, it forcibly schedules *CurrentOperation* in *CurrentCycle* and removes the operation in *UnscheduleSet* from the

Table 2: Schedulers and associated specialized Priority, Cycle and Unschedule classes

Scheduler	Priority	Cycle	Unschedule
`CycleScheduler`	`PriorityCycle`	`CycleEtimeOnly`	`UnscheduleNever`
`ListScheduler`	`PriorityDependence`	`CycleSequential`	`UnscheduleNever`
`OperBTScheduler`	`PriorityStatic`	`CycleSequential`	`UnscheduleAlways`
`ListBTScheduler`	`PriorityDependence`	`CycleSequential`	`UnscheduleSometimes`

partially generated schedule. If an operation cannot be scheduled in any of the valid cycles, and if `Unschedule.getForceCycle()` returns a non-`NULL` *ForceCycle*, the scheduler removes all conflicting operations from the partial schedule and forcibly schedules *CurrentOperation* in *ForceCycle*. Otherwise (if `Unschedule.getForceCycle()` returns `NULL`), the Scheduler invokes `Priority.push()` to push the *CurrentOperation* back into the unscheduled set of operations.

The three schedulers that we discussed before, the `CycleScheduler`, `ListScheduler`, `OperBTScheduler`, and `ListBTScheduler` are all implemented using the same `Schedule` class and surrounding infrastructure. The only difference is that when we construct each of the `Schedulers`, we use certain combinations of specialized versions of the `Priority`, `Cycle` and `Unschedule` classes. Table 2 shows the specialized classes that are used to implement a particular scheduler. We emphasize that the main scheduling loop that implements all four schedulers is identical.

6 Experimental evaluation

The schedulers were evaluated on a set of Unix benchmarks, viz. grep, cmp, eqn, qsort, tbl, wc, yacc, cccp. A 111L3 processor configuration with a branch latency of three and one each of integer, floating and memory units was used. The **OperBT** scheduler schedules each operation almost twice (1.95) compared to once for the **Cycle** scheduler, each operation on the average being scheduled 1.32 times normally and 0.63 times otherwise (displace or force). On approximately half the forcible scheduling steps, only one operation is unscheduled though in a few cases seven operations are unscheduled. Due to the fairly large number of unscheduling steps, the **OperBT** scheduler is substantially slower than the non-backtracking **Cycle** and **List** schedulers. The **ListBT** scheduler backtracks less often and runs almost as fast as the **List** scheduler. The **ListBT** scheduler runs even faster than the **Cycle** scheduler, because the **ListBT** scheduler always schedules the `CurrentOperation` it selects, whereas the **Cycle** scheduler may repeatedly push the `CurrentOperation` back to the `ReadyList` if resources are not available in the `CurrentCycle`.

Both backtracking schedulers generate similar quality schedules, with the **OperBT** scheduler only slightly better than the **List** scheduler. The improvement in estimated dynamic cycles using the backtracking schedulers averages around 2.2%. The percentage improvements in overall dynamic cycles are limited by the fact that empty delay slots account for a small fraction of overall cycles and the backtracking schedulers are targeted at filling these delay slots. A more comprehensive evaluation of the schedulers is currently underway in collaboration with Professor Waleed Meleis and graduate student, Ivan Baev, from Northeastern University, Boston.

7 Related Work

The early work on parallel instruction scheduling for processors was carried out in the context of microcode compaction [9-11]. The pre-scheduling steps such as the construction of dependence graphs are also examined in [12]. A reservation table, which was originally developed in the context of designing hardware pipelines [13], represents resource constraints during scheduling. Each row of a reservation table represents a particular cycle and each column a particular resource. A cell in a reservation table records the usage of the corresponding resource in a particular cycle. Eichenberger and Davidson describe efficient methods for representing resource constraints during scheduling [14].

Heuristics for scheduling instructions in a basic block for pipelined processors [15, 16] and superscalar processors [17] typically rely on ordering the operations in a block based on priority functions and scheduling the operations one by one as in **Cycle** or **List** scheduling. Approaches for basic block scheduling using optimization techniques such as finite state automata have been studied [18]. The joint optimization of scheduling and register allocation is addressed by [19] and [20]. General backtracking algorithms for constraint satisfaction problems have been evaluated [21].

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 [8, 22, 23]. A major concern is the complexity of inserting compensation code in the side entries/exits due to code motion between basic blocks in the global scheduling region. Global schedulers restrict the scheduling regions to reduce this complexity [24-26]. The schedulers described in this report use the superblock [27] [28] and hyperblock [29, 30] as the global scheduling region. Wavefront scheduling uses a region consisting of a tree of basic blocks [31].

Meld scheduling extends the scope of the optimization without increasing the complexity associated with code motion [32, 33]. Latency constraints are allowed to propagate outside the scheduling region, thereby giving some of the benefits of increasing the scheduling regions size

Programs spend a significant fraction of the total execution time in loops and special scheduling techniques have been developed for loops [34] [35]. The Cydrome Cydra 5 compiler uses modulo scheduling to schedule loops [36] [37]. Rau developed the iterative modulo scheduler, which backtracks in a manner similar to the **ListBT** scheduler [4, 5]. Modulo schedulers attempt to schedule for a particular iteration interval (II) and typically increase the II, if it is unable to generate a schedule within the prescribed compile time budget. Any schedule that meets a particular II are equivalent to the first order. These unique characteristics of modulo scheduling give rise to a different set of algorithmic choices. Firstly, 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** are guaranteed to never revisit the same partial schedule and never get into a repetitive orbit. This guarantee is essential because there is no option similar to increasing the II and trying to schedule again. Secondly, the iterative modulo scheduler must account for a cyclic dependence graph, which may constrain *LateCycle* even when no successor from the same loop iteration has been scheduled. The **OperBT** and **ListBT** scheduler does not face such constraints. Thirdly, the schedule length is the primary optimization metric in acyclic scheduling as opposed to a secondary metric in modulo scheduling. In order to optimize schedule lengths, **ListBT** and **OperBT** are allowed to unschedule operations in any cycle ranging from *EarlyCycle* through *LateCycle* (not necessarily at the *ForceCycle*), provided the operation has higher priority than the operations in *UnscheduleSet*.

8 Conclusions

This report 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 the familiar **Cycle** and **List** schedulers and describe the scheduling infrastructure. We then 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 picks operations primarily in dependence order and therefore greatly reduces the need for backtracking. Preliminary experiments demonstrate that both backtracking schedulers successfully fill a significant fraction of branch delay slots, providing an increase in schedule quality 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 author thanks all the members of the Compiler and Architecture Research (CAR) group at Hewlett-Packard Laboratories who were engaged in the Elcor compiler project. Waleed Meleis and Ivan Baev provided useful feedback and comments on earlier versions of this report.

References

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

[2] S.G. Abraham and S.A. Mahlke, "Automatic and efficient evaluation of memory hierarchies for embedded systems," Proc. Int. Symp. Microarchitecture, pp. 114-125, Haifa, Israel, Nov. 1999.

[3] S.G. Abraham and S.A. Mahlke, “Automatic and efficient evaluation of memory hierarchies for embedded systems,” Technical Report HPL-1999-132, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techreports/1999/HPL-1999-132.pdf` Nov. 1999.

[4] B.R. Rau, “Iterative modulo scheduling,” HPL-94-115, Hewlett Packard Laboratories, `http://www.hpl.hp.com/techreports/94/HPL-94-115.html` Nov. 1995.

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

[6] 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).

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

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

[9] S. Davidson, et al., “Some experiments in local microcode compaction for horizontal machines,” *IEEE Trans. Computers,* vol. C-30, no. 7, pp. 460-477, July 1981.

[10] B.R. Rau and J.A. Fisher, “Instruction-level parallel processing: History, overview and perspective,” Technical Report HPL-92-132, Hewlett Packard Laboratories, Oct. 1992.

[11] B.R. Rau and J.A. Fisher, “Instruction level parallel processing: History, overview and perspective,” *J. Supercomputing,* vol. 7, no. 1, pp. 9-50, May 1993.

[12] M. Smotherman, et al., “Efficient DAG construction and heuristic calculation for instruction scheduling,” Proc. Ann. Symp. Microarchitecture, pp. 93-102, 1991.

[13] E.S. Davidson, et al., “Effective control for pipelined computers,” Proc. COMPCON '75, pp. 181-184, San Francisco, Feb. 1975.

[14] A.E. Eichenberger and E.S. Davidson, “Reduced multipipeline machine description that preserves scheduling constraints,” Proc. ACM SIGPLAN Symp. Prog. Lang. Des. & Impl. (PLDI), pp. 12-22, May 1996.

[15] 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.

[16] J.L. Hennessy and T. Gross, “Postpass code optimization of pipeline constraints,” *ACM Trans. Prog. Lang. & Sys.,* vol. 5, no. 3, pp. 422-448, July 1983.

[17] 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.

[18] V. Bala and N. Rubin, “Efficient instruction scheduling using finite state automata,” *Int. J. Parallel Programming,* vol. 25, no. 2, pp. 53-82, Apr. 1997.

[19] J.R. Goodman and W.C. Hsu, “Code scheduling and register allocation in large basic blocks,” Proc. ACM Int. Conf. Supercomputing, pp. 442-452, 1988.

[20] S.S. Pinter, “Register allocation with instruction scheduling: A new approach,” Proc. ACM SIGPLAN Conf. Prog. Lang. Des. & Impl. (PLDI), pp. 248-257, June 1993.

[21] G. Kondrak and P.v. Beek, “A theoretical evaluation of selected backtracking algorithms,” *Artificial Intelligence,* vol. 89, no. 1-2, pp. 365-387, Jan. 1997.

[22] J.A. Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE Trans. Computers,* vol. C-30, no. 7, pp. 478-490, July 1981.

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

[24] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Prog. Lang. & Sys.*, vol. 9, no. 3, pp. 319-349, July 1987.

[25] 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.

[26] 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.

[27] P.P. Chang, S.A. Mahlke, and W.M. Hwu, "Using profile information to assist classis code optimizations," *Software Practice & Experience*, vol. 21, no. 12, pp. 1301-1321, Dec. 1991.

[28] 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.

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

[30] S.A. Mahlke, et al., "Sentinel scheduling: A model for compiler controlled speculative execution," *ACM Trans. Comp. Sys.*, vol. 11, no. 4, pp. 376-408, Nov. 1993.

[31] 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.

[32] S.G. Abraham, V. Kathail, and B.L. Dietrich, "Meld scheduling: A technique for relaxing scheduling constraints," Technical Report HPL-97-39, Hewlett Packard Laboratories, `http://www.hpl.hp.com/techreports/97/HPL-97-39.html` Feb. 1997.

[33] 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.

[34] A.E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," *IEEE Computer*, vol. 14, no. 9, pp. 18-27, 1981.

[35] B.R. Rau and C.D. Glaesar, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," Proc. 14th Ann. Workshop on Microprogramming, pp. 183-198, Oct. 1981.

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

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