ABSTRACT

Title of dissertation: VLIW INSTRUCTION SCHEDULING FOR REDUCED CODE SIZE

Steve Haga, Doctor of Philosophy, 2005

Dissertation directed by: Professor Rajeev Barua
Dept. of Electrical and Computer Engineering

Code size is important to the cost of embedded systems. Although VLIW architectures are popular for embedded systems, they impose constraints on instruction placement that make it difficult to find a compact schedule. Existing VLIW instruction scheduling methods primarily target run-time but not code size. The usual approach has two components. First, methods such as trace scheduling provide a mechanism to correctly move instructions across basic blocks. Second, the instructions within a trace are scheduled, perhaps moving instructions across blocks. Because run-time is the only consideration, this approach increases code size by inserting compensation code. Methods such as superblocking increase the size even further by duplicating code.

We present a compiler method for instruction scheduling that, for the first time, uses the power of across-block scheduling methods such as trace scheduling to reduce code size as well as run-time. For a certain class of VLIWs, we show that trace scheduling, previously synonymous with increased code size, can in fact reduce it. Our within-trace scheduler uses a cost-model driven, back-tracking approach.

Starting with an optimal, exponential-time algorithm, branch-and-bound techniques and non-optimal heuristics reduce the compile time to within a factor of 2 of the original, on average. The code size for our benchmarks is reduced by 16.3% versus the best existing across-block scheduler, while being within 0.8% of its run-time, on a 6-wide VLIW. For a 3-wide VLIW, code size improves by 14.7%, with the same 0.8% run-time cost. Thus, the code size improvements are fairly stable across VLIW widths.

We further explore the impact of our techniques on machines with predication support or small I-cache sizes. In the process, we present a novel predication analysis of general applicability. If predication is present, the code size improves to 16.6%. In addition, for machines with small I-caches, the reduced code size of our approach tends to yield better cache hit rates. We find that, although this effect is modest, the performance improvement more than offsets the run-time costs of our method. Therefore, on machines with small I-caches, our code size improvements are achievable at no run-time cost.

VLIW INSTRUCTION SCHEDULING FOR REDUCED CODE SIZE

by

Steve Haga

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

ACKNOWLEDGMENTS

First and foremost, I must acknowledge my Heavenly Father and his Son, Jesus Christ. I know that is only through his blessing and provision that I could complete this dissertation, so I dedicate it to him. I thank him for teaching me that the foundation of wisdom is the fear of the LORD. May my dissertation in some way glorify him, as all things must, because only he is worthy to receive glory and honor and power, for he created all things, and by his will they were created and have their being. (Rev 4:11)

I would also like to acknowledge those who have helped me to complete this dissertation. My advisor, Dr. Rajeev Barua, has been an outstanding teacher. He taught me every aspect of successful research, trained me in technical writing, and provided a great deal of advice for my future. At many difficult points in the research, he had keen insight into how to proceed. He has also been very understanding and supportive of me in practical ways. I am very grateful for his support and guidance.

I also wish to bring to attention the contributions of a variety of summer research interns. Nghi Nguyen and Andrew Webber assisted in the implementation of the predication mechanism, and aspects of trace selection. Itai Katz and Kaushik Veeraraghavan assisted in implementing profiling into the compiler, along with some additional work in trace selection. Adam Archer assisted at an early stage of the research, in the selection of the sgicc compiler.

Several graduate students were also very helpful. First of all is Yi Zhang, who contributed to debugging the code, as well as in discussions on the algorithm. In addition, Angelo Dominguez supplied invaluable support as our group's network administrator. He helped to get the compiler infrastructure running and to troubleshoot many problems over the years. Sumesh Kumaran, T. Vinod Gupta, and Tom Carley also contributed ideas during our group meetings.

I wish to thank my family, as well. My wife, Allison Haga, prayed for me and encouraged me to work hard. She took care of many chores so that I could concentrate on my research. I also must thank my parents, Donald and Jo Marie Haga, who made many sacrifices for my education, and always lovingly provided my family with many things. In addition, I thank my in-laws, Li Kwan Chuen and Dr. Cheung Sau Chun, for their support.

I lastly recognize my church members, who have prayed for me and my studies for many years. In particular, I thank missionaries Hannah Ku, Esther Lee, and Jacob Lee. As God has answered their prayers for me, may he also pour out many blessings upon them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Very-long Instruction Word architectures (VLIWs) are the predominant design in embedded systems for exploiting instruction-level parallelism. In a traditional VLIW, the compiler must identify independent instructions for execution in parallel, and then place them together into fixed-length instructions groups called *long-words*. This situation contrasts with desktop processors where superscalar architectures dominate – in a superscalar the hardware, not the compiler, discovers and schedules parallelism.

VLIWs are used in embedded systems because they offer similar performance to superscalars at a lower cost [7, 8]. The drawback of VLIWs is that they rely more heavily upon good compiler technology. Such technology has slowly improved over the last two decades. This paper presents an instruction scheduling method for VLIWs that specifically targets embedded systems – aiming to reduce not just run-time but also code size, which is very important for embedded systems.

To understand how to improve VLIW instruction scheduling, we must first understand how current approaches work. VLIW instruction schedulers consist of two phases. In the first phase, trace scheduling [2] or its later improvements [4, 9] derive *traces* – sequences of basic blocks that are likely to follow one after another with high probability. More importantly, they provide the mechanisms to move instructions from one basic block to another within the trace, allowing for scheduling flexibility. This first phase decides which instructions can legally move across basic block boundaries, but it does not move any instructions. Movement decisions are made by the second phase of scheduling, which assigns the instructions within each trace to the available issue slots of the VLIW, moving instructions across basic blocks if it improves the schedule. This second phase also considers the instruction-type restrictions per VLIW issue slot, the instruction dependencies within the trace, and the allowed across-basic-block instruction moves. Producing a good schedule is an NP-complete problem [10] and thus heuristics are needed. Existing second-phase methods for scheduling within a trace include list scheduling [10], finite state automata [1] or dynamic programming [11]. In this phase, the trace is treated as though it were a simple basic block. Therefore these methods are known as basic-block scheduling algorithms rather than as trace-scheduling ones. Most of today's basic-block schedulers use one-pass (greedy) heuristics, and yet are fairly effective at finding a fast schedule (See Chapter 10).

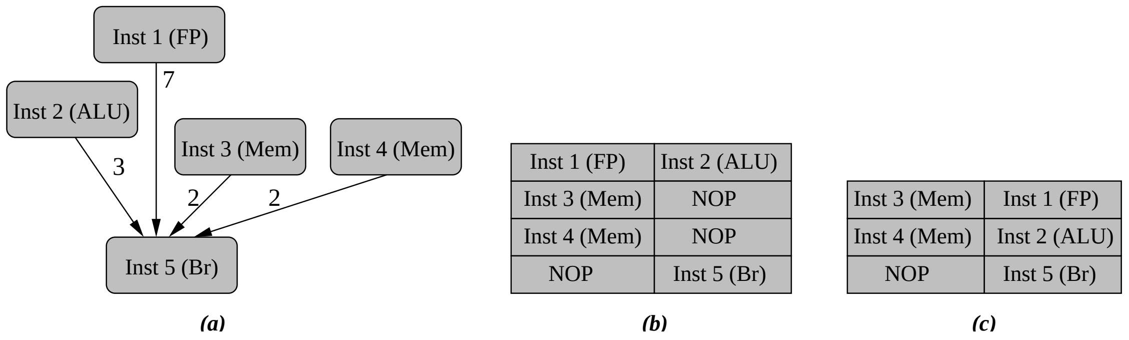
A significant drawback of existing scheduling methods is that they only aim to reduce the run-time of the executable; they ignore code size when scheduling instructions. This is not surprising since most existing methods [10, 1, 11] were either designed for older VLIWs which require explicit long-words to perform stalls, or for desktop processors where the presence of virtual memory and large hard disks makes reducing the code size less important. In embedded systems, however, reducing code size is extremely important [12] for reasons of cost reduction – code is usually stored in Read-Only-Memory (ROM), and smaller code implies that less

ROM is needed, lowering its dollar cost, as well as power consumption and access time. Existing across-basic-block schemes are likely to further increase the code size versus a simple basic block scheduler that does not move instructions across blocks at all, because moving instructions across block boundaries will often violate program correctness unless extra instructions, called *compensation code*, are inserted outside of the trace (as in Chapter 7) [2, 4].

1.1 A motivational example

Here we present an example of how a current-day scheduler can produce a schedule with minimum run-time, but non-minimum code size. This example is important since it underscores the difference between a method that successfully produces a minimum run-time solution and one that also yields the smallest code size. Figure 1.1(a) presents the dataflow graph (DFG) for the dependencies of a sample basic block. The nodes are instructions; the edges are dependencies between instructions; and the numbers attached to the edges are their latencies. To simplify the discussion, let us try to schedule this DFG on a basic, 2-wide VLIW. Since VLIWs typically have fewer copies of a functional unit than the VLIW width, we will assume that the machine has one memory unit, and restricts memory instructions to only be schedulable in the first slot.

Figure 1.1(b) shows the run-time schedule that results from list scheduling [10]. Although only four long-words are scheduled in 1.1(b), the run-time is eight cycles, which is, incidentally, optimal in this case, since the critical path in the DFG is also



(b)

Inst 1 (FP)	Inst 2 (ALU)
Inst 3 (Mem)	NOP
Inst 4 (Mem)	NOP
NOP	Inst 5 (Br)

(c)

Inst 3 (Mem)	Inst 1 (FP)
Inst 4 (Mem)	Inst 2 (ALU)
NOP	Inst 5 (Br)

Figure 1.1: VLIW scheduling example. (a) DFG; (b) result of list scheduling, assuming that only one slot can perform memory operations; (c) an optimal schedule.

eight cycles. (For simplicity of presentation, this example assumes the presence of hardware stalls. Similar examples can be easily constructed for multi-NOP systems.) While the schedule of 1.1(b) has achieved the optimal run-time, its code size is not minimum. Figure 1.1(c) has the same run-time (eight cycles), but a smaller code size (three long-words instead of four). This example shows how current methods that optimize for run-time alone may not produce minimum code-size solutions. Fortunately, it also shows that minimizing code size need not sacrifice run-time – our scheduler minimizes code size but retains the same run-time. In fact, there typically exists an entire family of optimal-run-time schedules for a basic block, and these schedules may have different code sizes. This example uses list scheduling because it is easy to understand. Better methods than list scheduling may occasionally find a faster schedule, but they do no better in terms of code size because *all existing methods only consider run-time,* so that Figures 1.1(b) and 1.1(c) *appear equally good.*

Delving deeper, this example illustrates a fundamental drawback of all greedy schedulers in use today. In Figure 1.1(b), the greedy scheduler, choosing instructions based on the earliest deadline, places instructions #1 and #2 in the first long-word. This is optimal so far, but *it does not consider the impact of the current scheduling choice upon the remaining code.* As a result of this selection, there will be contention for the memory slot on later cycles. Only a backtracking scheduler, such as ours, can avoid this type of mistake. By exploring alternative schedules, it effectively undoes the mistake of the greedy assignment.

1.2 Our proposed approach

This paper presents a scheduling method for reducing code size while maintaining the run-time of existing approaches. It runs in two phases: an *across-block* analyzer which discovers which instruction moves will likely increase code size, followed by a *within-trace* scheduler. These are outlined below.

First, the across-block analyzer does not actually schedule any instructions, but calculates which subset of instruction moves are likely to increase code size, and which are not likely. Those moves that are expected to increase code size are then marked as 'disallowed' by introducing appropriate new control dependencies.

Computing the subset of moves that increases code size is somewhat involved, however. An instruction move does not increase code size if no compensation code is needed, or if the compensation code can replace an existing NOP in the schedule. The first case – no compensation code needed – is detected by looking at the liveness properties of the target register of the moved instruction, as will be described in

Chapter 7. The second case – when compensation code fits into existing NOPs – is harder to compute since the scheduling has not been done yet, and thus the location of NOPs is not known! To overcome this problem, our method attempts to order traces in such a way that the traces containing the compensation blocks will be scheduled before the trace that attempts to copy into them. This is not always possible, and so the side trace is sometimes tentatively scheduled, without allowing compensation code. Then, the NOP locations of this tentative schedule are used to approximately estimate whether a particular instruction move will result in code growth or not. An important result of our work is that *some across-block code motions may actually reduce code size*; this happens if both the instruction and its compensation code fill existing NOPs.

Second, scheduling of instructions within traces is done. Such scheduling considers each trace as if it were one giant basic block. During this phase, all constraints on instruction movement across blocks, computed in the first phase, are respected so that code-size-increasing moves are disallowed. The traces are scheduled in decreasing order of their execution frequency, thereby giving preference to the compensation code from more frequent traces to fit in the NOPs of other traces. At this time, any traces that were tentatively scheduled without allowing code motion will also now be rescheduled with code motion.

The within-trace scheduler is built as follows. Since our problem is more difficult than the one solved by existing trace schedulers - we optimize for *two* parameters (code size and run-time), where as current approaches optimize for only one (run-time) - adapting an existing greedy scheduler to reduce code size is unlikely to be effective. Instead, we go further and develop a back-tracking scheduler that specifically optimizes for code size and run-time. A back-tracking scheduler can undo early scheduling decisions if those decisions are found to be mistakes in the light of later instructions – something a greedy scheduler cannot do. Our back-tracking scheduler is based on an optimal exhaustive search algorithm which is infeasible because of its exponential compile-time, but which is made feasible through pruning techniques and non-optimal heuristics to restrict the compile-time increase to a user-controlled factor of the original compile-time. Schedules that take longer will simply quit. Therefore, an upper bound on the compile time is derived by assuming that every trace uses its full allotted time and then times out without finishing. We have chosen the current time-out value so that this is approximately a factor of 6 greater than the original compile time. This compile-time bound is based on the trace sizes and dependencies, so is not the same value for all programs. *Yet it is a constant for any particular program.* That constant value never exceeds a factor of 10 for any of our benchmarks.

It is important to note that the *actual* compile-time is much smaller, being, on average, a factor of two greater than the original. The factor-of- six upper-bound is the theoretical maximum if every schedule in the entire program were to time-out; in reality, only a few traces time-out, while most either finish searching or find a minimal solution before the time-out heuristic is needed.

The within-trace scheduler only searches among solutions that have minimum run-time, and among those chooses the schedule of smallest code size. Previously proposed schedulers make no effort to reduce the code size, perhaps because of a widespread belief that reducing run-time will automatically reduce code size. This is only approximately true, however – we show that it is possible to reduce code size further without any sacrifice in run-time, because two solutions of minimum run-time may have differing code sizes.

Although the above two-phase method is complete and reduces code size, it can be improved further by changing the optimization criteria for traces that are either very frequent or very infrequent. While the above scheduler considers both run-time and code size to be important, this is not necessarily true at the extremes. In particular, for traces that are very frequent, we modify our method to optimize for run-time alone. Thus, for the sake of the run-time of frequent traces, we may allow compensation code that increases code size. Conversely, for traces that are very infrequent, our method is modified to optimize for code size alone, even at the cost of run-time, since the run-time of these blocks is of little importance. In this way, our method can improve code size further and yet sacrifice little on run-time versus the best existing methods that are optimized only for run-time. We refer to this as our *hybrid* algorithm, because it uses different algorithms for different traces.

As an add-on feature to our method, we also explore how predication can improve code motion opportunities. Predication is a hardware mechanism that allows dynamic control of whether a fetched instructions will actually be executed. If an instructions predicate bit is 0, the instruction is skipped when its long-word is executed. For our purposes, we use the predication mechanism to allow compensation code to be inserted into locations where it otherwise would not be legal. In the process, we develop a new method of predication analysis which is more broadly applicable to other compiler optimizations.

Since our techniques improve code size, they will also improve the performance of the instruction cache (I-cache). For embedded machines with limited I-caches, or for programs with poor I-cache performance, this effect may be marginally significant; our studies indicate possible improvements of around 1%. Although small in itself, this positive side effect of our methods demonstrates that a slight performance improvement is possible in addition to the code size advantages of our approach.

The intellectual novelty of our scheme is seen in the following four new contributions. First, although the idea of using a back-tracking scheduler is not new (see related work), we are the first to develop a back-tracking technique targeted for code size, and to develop a series of innovative pruning techniques unique to a search for a minimum code size solution. Second, using across-block motion to reduce code size is different than existing across-block approaches that generally increase code size. To this end, our method constrains the within-trace scheduler to not move certain instructions across basic blocks if that move would likely increase code size. This is unlike all existing schedulers which do not need to place additional constraints on movement since the within-trace scheduler alone decides whether it is profitable to move instructions, based solely on run-time. Third, our method is unique in that it orders the scheduling of traces in such a way as to reduce code migration into unscheduled traces, while at the same time allowing a mechanism for preliminarily scheduling traces to estimate which instruction moves are likely to increase code size, before performing within-trace scheduling. Such a preliminary schedule is not needed when optimizing for run-time alone. Fourth, our method is unique in targeting different objectives for traces of different frequencies – code size only for infrequent traces; run-time only for frequent traces; and both code size and run-time for traces of intermediate frequency. Such customization for multiple objectives is not needed in traditional schedulers since they have only one primary objective of reducing run-time.

Our experimental setup, platform and benchmarks are described in detail in Chapter 10. A summary of our results are as follows. Compared to the across-block scheduler of [1] augmented with the code size reduction techniques of [3], our scheduler reduces the code size by 16.3% and our run-time is only 0.82% slower. This code size improvement can be divided among three sources. First, using only our within-trace scheduler improves code size by 8.1%. Second, adding the across-block code movement restrictions improves code size by another 4.3%. Third, using a hybrid strategy that includes optimizing for code size only and ignoring run-time for the least frequent blocks improves code size by yet another 3.9%. Thus the total code size improvement is the sum of the three: 16.3%.

An outline of the dissertation is as follows. Chapter 2 describes related work. Chapter 3 describes which VLIWs can benefit from our method. Chapters 4 through 6 describe our backtracking scheduler as it applies to basic blocks, without considering special issues relating to moving instructions across basic blocks. (It is still called a trace scheduler, however, since it will be used for traces later chapters. Even when run on individual basic blocks, it is not wrong refer to the methods as a trace scheduler, since traces can be single basic blocks.) Chapter 4 describes the basic optimal algorithm; Chapter 5 considers branch and bound techniques to reduce compile-time; and Chapter 6 lists non-optimal heuristics to further reduce compile time. Chapter 7 then explores modifying this backtracking scheduler so that it can accommodate traces spanning several basic blocks. Chapter 8 shows how to do better by changing the optimization criteria at the extremes. Chapter 9 considers extending our algorithm to exploit predicate information, when present. Chapter 10 lists experimental results. Chapter 11 provides conclusions. Also included is Appendix A, which derives our Finite State Automata based on the specific details of our test machine.

Chapter 2

Related Work

Instruction scheduling algorithms can be divided into 1) those methods that permit instructions to move *across* basic blocks based on liveness analysis along the control flow graph, and 2) those methods that actually schedule the instructions *within* a trace based on the data dependencies and slot restrictions. Traditionally, these two types of algorithms have been used in tandem to improve the run-time of critical execution-paths within the program. In contrast, our method targets both of these related problems from the viewpoint of reducing code size as well.

In this chapter, Section 2.1, we discuss current methods for scheduling instructions across basic blocks. Section 2.2 explores existing research in scheduling instructions within a trace. Section 2.3 compares our approach to related work in compiler-driven code compression. Section 2.4 considers related work in hardware methods used in current-day processors.

2.1 Across-block scheduling

In this section, we begin by considering the motivations for across-block scheduling. We then explore a variety of existing across-block methods: *traces*[2], *superblocks*[4], *hyperblocks*[9], and *wavefronts*[27].

Motivation Allowing instructions to schedule across basic blocks provides two key areas for performance improvement. First, by optimizing portions of code that are likely to run together, the scheduler can optimize any data dependencies that exist between instructions in different basic blocks. To see this, we point out that all across-block algorithms aim to combine instructions from multiple basic blocks into one structure, which is then scheduled by any within-trace scheduler *exactly as if this structure were a single basic block.* Second, by converting a series of smaller blocks into one large block, we create more scheduling flexibility and may uncover additional parallelism that will allow the VLIW slots to be more fully utilized along the likely paths. We will see, however, that such code movement often requires instructions to be duplicated, so that across-block methods typically increase the code size.

Trace scheduling [2]. We now define the key terms and concepts of trace scheduling, since it is the basis of most more-advanced methods for across-blocks code motion, including ours. To begin, the *trace*, is formally defined as a set of basic blocks with a high probability of following one after another during the execution of the program. These block execution probabilities are found through profiling. Given a trace, those basic blocks which it contains may be referred to as *trace blocks* and all other block may be termed as *off-trace blocks.* Therefore, a trace containing n basic blocks may be expressed as an ordered set of trace blocks: $(b_1, b_2, ..., b_n)$. These blocks need not follow one another sequentially in the code (*i.e.,* they do not have to all be fall-through blocks). A *side exit* is defined as a control flow edge from any trace block, b_i to any basic block *other than* b_{i+1}, where $i < n$. We note that exits from the end of the trace (corresponding to exits from b_n) are not side exits, by this definition. Further, we see that even an edge to another trace block, $b_j, where j \neq i+1, is$ a side exit. For such a trace block, b_j, we can in fact assert the stronger condition that $j > i + 1$, because traces may not contain internal cycles. A *side entry* is similarly defined as a control flow edge to a trace block, b_i from any block other than b_{i-1}, where $i > 1$. The set of side exits and set entries are referred to collectively as the *out-of-trace edges.* Trace scheduling may also require *compensation code.* As previously observed, across block methods are used in tandem with a within-trace scheduler; when instructions are moved across basic block boundaries by the within trace scheduler, compensation code must be inserted into the off-trace paths. Compensation code is described in detail in Chapter 7. For the moment, it is sufficient to understand that the term refers to extra copies of moved instructions which are needed to preserve correct program behavior, but which also tends to increase code size.

Since compensation code increases code size, *dominator-path scheduling* [5] proposes limiting code motion across blocks to only those movements that do not require compensation. A position in a trace is said to *dominate* an instruction if, for every path to that instruction, none of the registers used by the instruction are modified. Therefore, instructions can be safely moved across basic block boundaries, so long as they do not cross a dominating instruction. Imposing such a restriction on code motion may reduce the run-time benefit of trace scheduling, however. In [5], a modest performance improvements is obtained versus single-block scheduling, but the results are not compared against unrestricted trace scheduling.

Freudenberger et. al. [3] present an approach to reduce the compensation code created by trace scheduling by avoiding multiple copies of the same compensation code. [3] is an improvement upon *dominator-path scheduling* [5]. In [3], specific cases of compensation code are examined, in order to reduce their size. For instance, when an instruction is moved above a multiple-entry point, the compensation blocks for all of the side entries can be merged into a single compensation block. As another example, compensation copies are avoided along rejoin paths, whenever the copy can be shown to be redundant; a rejoin path is any sequences of control edges where the first edge in the sequence is a side exit from the current trace, and the final edge is side entry back into the trace. In addition to such special-case optimizations, [3] also prevents moving instructions below splits. They note that, though this reduces the available parallelism, it rarely affects performance. We find that, when considering code size as well as run-time, however, this restriction will be too costly, so we *do allow* downward motion, even in our comparison algorithm that is based on [3]. [3] cannot be compared in any meaningful way to our algorithm for two reasons. First, we wish to not only *avoid* the code size cost of trace scheduling, but also to leverage trace scheduling to *reduce* code size, something that is impossible without the within-trace scheduler that we are also proposing. Second, and most importantly, we are not competing against [3]. We have implemented approaches similar to [3] into *both* our method and our comparison algorithm, because [3] represents one of the best existing efforts at reducing the code size increase of across-block instruction scheduling. Therefore, the across-block methods that we propose are built on top of [3]. In this way, our results show the additional benefits of our method beyond this prior work.

Superblock scheduling [4]. Other across-block schedulers, besides trace scheduling, exist; superblocking is currently the most popular. A superblock is defined as a set of blocks in which control may only enter from the top, but may exit from multiple points. Although there may be side exits, the superblock still must have a high probability of executing all of its basic blocks, just as was required for trace blocks. In fact, superblocks are created from traces. The difference with a trace is that it may contain side entry locations. Therefore, the superblock must modify the control flow graph so that these side entries point to somewhere else, through a process called *tail duplication*. A tail is defined as a subset of trace blocks: $(b_i, b_{i+1}, ...b_n)$, where $i > 1$. A side entry into the superblock can be removed by the process of: 1) creating a new trace consisting of the tail portion of the superblock, below the side entry point, 2) modifying the destination of the side entry edge to point to the tail copy, rather than to the original trace block, and 3) ensuring that any branches from the tail copy to other trace blocks still point to the original blocks and not their copies. By repeating this process for all side entry positions, a trace is converted into a superblock. Then each of the tail copies may be similarly converted into their own superblocks, through additional tail duplication. It is always possible to convert a trace into a set of superblocks by this approach.

There are two advantages to the superblock. First, the probability of executing all of blocks in the trace increases, because the different side entry probabilities are no longer a concern. Second, since there are no side entries, there will be fewer restrictions on code motion. In fact, if a block, b_i, does not have a side exit, then it can merge with b_{i+1}, once b_{i+1}'s side entry has been removed through superblocking. Since b_i and b_{i+1} have effectively become one basic block (*i.e.,* . there is no longer any control flow between them), there will clearly be no across-block-code-motion restrictions.

While superblocking improves performance even more than trace scheduling, it also increasing the code size far more than trace scheduling. Superblocking still suffers from compensation code, but this cost is dwarfed by the new cost of the tail duplications. For this reason, superblocking is unsuited for embedded systems with strict code-size constraints.

Hyperblock scheduling [9] Another scheduling technique that builds upon the superblock is the hyperblock. Hyperblocks are predicated superblocks. The hyperblock is a way to incorporate *if conversion* (discussed in Chapter 9) with the superblock. Therefore, the hyperblock cannot be used for our purposes, because it suffers from all of the code-size difficulties of the superblock, and, in addition, because it is limited to machines with predication.

As with hyperblocking, we have also considered how our across-block algorithm may be modified to take advantage of predication. For clarity of presentation, however, we defer our discussion of the related work on predicated scheduling until Chapter 9. Since these benefits do not apply to all machines and are not central to the understanding of our more general method, this material is best covered in its own independent chapter, after the basic methods have been fully presented.

Wavefront scheduling [27] There are other alternatives to trace scheduling, such as wavefront scheduling [27]. In this method, the data dependencies among all of the instructions in the entire function are considered at once. At any point in the assignment process, the instructions can be partitioned into three groups: those instructions that have been scheduled, those that are ready to schedule, and those that have not scheduled. Instructions are chosen from among those that are ready to schedule, referred to as the wavefront, based on heuristics. The advantage of this technique is that it does not suffer from as much tail duplication as superblocking, because only useful instructions are placed in the tails. Such an approach still increase code size more than trace scheduling, however.

For these reasons we compare our results against the best trace scheduler in the literature that makes an attempt to reduce the code size increase from compensation code [3].

2.2 Within-trace scheduling

Once a trace is identified, its instructions are scheduled as if they were a single basic block, so we now explore the related work in within-*block* scheduling. We refer to these methods as within-*trace* schedulers, however, because we are technically using them to schedule traces and not basic blocks. Yet the distinction is only semantic, because no modification is required to use a within-block scheduler on a trace.

Concerning scheduling instructions within a trace, list scheduling, described in Chapter 1.1, has proven sufficient for superscalars – they have a re-order buffer to correct bad schedules, and have no constraints on instruction placement. The widespread use of VLIWs in embedded applications has motivated more advanced techniques that consider the resource constraints of the system.

One such technique by Bala et. al. [1] schedules instructions based on finite state automata (FSA) methods. The resource requirements of each instruction are represented as a bit-vector. If the AND-ing of two bit vectors yields an empty vector, then the two corresponding instructions do not share any resources, and may be scheduled together. Given that a particular set of instructions has been scheduled in a certain cycle, the OR-ing of their bit vectors represents the resources currently used. To construct the FSA, a bit vector of resources used by the current set of instructions represents a state, and choosing to schedule an additional instruction on this cycle represents a transition to a new state. Therefore, legal instruction schedules can be identified as the sentences in a language whose alphabet is given by the instruction set. This makes it easy to ask questions about whether new instructions will fit with others in the same long-word, and we employ this approach. The scheduling of instructions within a trace is performed in [1] using a straightforward greedy approach: instructions are placed on a cycle until it is filled, and then the next cycle is scheduled. This approach also has application to scheduling into off-trace paths, as it is easy to identify whether an instruction can fit into an existing NOP slot. For reasons outlined in the results section, this approach, augmented with trace scheduling and the code size reducing techniques in [3], is used as our comparison in evaluating the benefit of our method.

A number of optimal (for run-time) instruction schedulers have been proposed on a variety of systems [28, 11, 30, 31], using backtracking or ILP formulations. But as these approaches also only target run-time, they perform similarly to [1] in regards to code size. Still, scheduling by branch and bound, coupled with heuristics, is not a new idea. For example [32] takes such an approach to scheduling instructions so as to reduce register spills, in embedded DSP microprocessors. (Our method occurs after register allocation and so is not concerned with spills.) [34] also applies branch and bound techniques to instruction scheduling. Their target, however, is to reduce run-time for a very unique VISC architecture. Other non-optimal approaches to instruction scheduling include genetic algorithms [35]. [35] targets a different problem of making a compiler retargetable to different *scalar* architectures. But, *much more importantly*, none of these proposed schedulers considers the problem we address: code size. Since these approaches only target run-time, they perform similarly to [1] in regards to code size. Ultimately, the novelty of our within-trace scheduling technique is the application of branch and bound techniques to code size, coupled with aggressive pruning strategies tailored to code size.

2.3 Existing compiler approaches to reducing code size

Many compiler optimizations have also been proposed for code size, some are implemented entirely in software, but others require (or at least benefit from) hardware support. Usually, the hardware support involves adding a new instruction to the ISA. Most compiler research for code size can be classified into one of three general categories: 1) methods that decrease code size by reducing the number of instructions in the binary executable, 2) methods that decrease code size through a dictionary-based interpreter of a reduced instruction format, and 3) methods that decrease code size by applying data compression (such as Huffman encoding) to infrequent code-regions.

First we consider optimizations that reduce the number of instructions in the compiled program. Many classical compiler optimizations fall into this category, such as dead-code elimination and common-sub-expression elimination [10]. In many cases, these optimizations are mostly employed because they also reduce the run-time, but [52] uses them to specifically reduce code size. Such optimizations solve orthogonal problems to the one that our present work considers. As a result, these techniques are not in direct competition with our approach; rather, the best code-size results are obtained by optimizing for all orthogonal problems. In fact, our compiler infrastructure already performs many of these classical optimizations.

Building on the concept of common sub-expression elimination, *procedural abstraction* [32] identifies common code fragments, then replaces each of these duplicated sequences with function calls to new functions that contain the original instruction sequences. This optimization can be thought of as the opposite of procedure inlining. Conceptually, these abstracted procedures may contain any number of basic blocks, and they may be called from different procedures. In many cases, however, implementations tend to have the granularity of simple code-segments located within one parent procedure, although [36] does extend this approach to inter-procedural analysis. Procedural abstraction, like the classical optimizations of the last paragraph, solves an orthogonal problem to the one explored by our method.

To reduce the overhead of the function calls to newly-abstracted procedures, [33] proposes adding a *call-dictionary* instruction to the ISA. This new instruction acts like a normal call instruction, except that: 1) the call address points to a special hardware dictionary rather than to main memory, and 2) the new call-dictionary instruction contains a second argument to indicate the number of instructions that are to be executed before returning back to the caller function. By using an argument to pass this instruction-count number, the abstracted procedure does not need a return instruction. In addition, this instruction-count argument allows for variable execution lengths: different call sites may execute different sub-sequences of the same abstracted procedure, being able to both start *and* stop at call-site specific addresses, should it prove beneficial to do so. Since call dictionaries are based on procedural abstraction, then as with the methods described above, call dictionaries tackle an orthogonal problem to the one that we address. Therefore, they may be used in conjunction with our scheme.

These call-dictionary instructions are further developed by [61] into *echo instructions*. Here, no special hardware dictionary is needed because the instruction sequence resides in instruction memory. Moreover, rather than creating a new procedure, one of the copies of this instruction sequence is left in its original location, while all other instances will contain "echo" instructions, which behave very similarly to call-dictionary instructions. As a further extension of this approach, [51] proposes *bit-masked echo instructions*. In bit-masked echo instructions, the instruction-count argument of the call-dictionary is replaced by a more flexible bit-mask that allows each call site to selectively execute a non-sequential subset of the echo instructions. In the end, code-size reductions of around 15% were observed for the Alpha processor. Being yet another form of procedural abstraction, echo instructions similarly target an orthogonal to the problem that we solve, and can be incorporated into a compiler together with our method.

In [12], code size is reduced for embedded systems that are programmed by block diagram languages. These languages are based on a model of computation with strong formal properties [57]. This method does not apply to common languages such as C.

The second category of code-size reducing compiler methods refer to those that use interpreted byte-codes, such as values stored in a run-time modifiable dictionary, to compress instructions. Unlike the methods of the first category, which only removed repeated instructions, these methods reduce the code size by expressing instructions into a smaller number of bits. When these instructions need to be executed, however, they must first be interpreted. There are two possible methods: 1) the compiler may insert an interpreter routine into the binary executable and then insert calls to that interpreter routine just prior to the location of the compressed code [45, 48], or 2) the hardware may provide support either to execute code in its un-interpreted form [52] or to allow a special instruction for directly manipulating the cache [50]. Typically, the hardware-based approaches to this problem are more common. The interpreted-compression scheme is similar to just-in-time compilation. In addition, since there is a run-time overhead associated with interpreting instructions, these methods are typically only used for infrequent code-regions. In comparison, our approach applies to all code regions – although there are optimizations for infrequent portions. Also, our approach has little run-time overhead (or, if desired, certain of our optimizations may be turned off to achieve *no* run-time overhead.)

The third category of code reducing methods is similar to the second, except that compression methods such as Huffman encoding are employed, rather than using interpreted instructions. Compared to category 2, these methods cannot interpret instructions as they are fetched. As a result, the entire compressed code region must first be decompressed before its execution can begin.

As one instance of a category 3 method, [53], proposes arithmetic encoding. Arithmetic encoding is a very powerful compression algorithm, but with an equally-high computational latency. As a result, [53] partitions the code into large chunks that are separately zipped so that the decompression cost is amortized, and so that costly transitions are infrequent. When control flow enters a compressed block, the entire block is decompressed and stored in the I-cache. Therefore, the optimal size of compressed regions is a trade-off between reducing transitions across regions and avoiding the overuse of the I cache. In comparing this approach to our method, we note that, as is the case in category 2, our method avoids the run-time cost of decompression, and also applies to all code regions. In addition, we note that the code produced by our method can be compressed, if so desired. In this way, the work of [53] is orthogonal to our problem, and both methods may be effectively used together.

Others have chosen to use less-powerful-but-faster compression methods, such as Huffman encoding. In [37], infrequent code regions are encoded using a light-weight form of Huffman encoding, and are decompressed on demand into a reusable buffer. Unlike many other decompression algorithms, this scheme has the advantage of not requiring any special hardware, so that it is generally applicable. In [37], a 13.7% improvement was achieved for alpha binaries at no run-time cost (because only very-infrequent code was compressed). In comparing to our approach, all the observations of the previous paragraph apply to [37] as well: our approach does not limit its application to infrequent blocks, and our method is orthogonal, so that the methods of [37] could be used to compress the code that our method produces.

Among these three groups, our method is closest to category 1, because we also remove instructions so as to achieve a smaller code (albeit that the instructions we remove are actually all NOPs). Yet, even compared to the methods of category 1, we use a quite different approach that is not based on Procedural Abstraction, but on compact instruction scheduling. Although it is difficult to compare compression sizes on different architectures – and especial between VLIWs and superscalars – we do note that our improvements are similar to those obtained by many of the above methods, but without requiring new hardware, as most existing approaches propose. More to the point, we note that, since our approach looks for code reduction from a different source than any of these previous methods, our approach can be used in conjunction with many of them, for an additional improvement.

2.4 Existing hardware approaches to reducing code size

In the previous section, one may observe that many compiler techniques require hardware support (often this support comes in the form of adding a single code-size reducing instruction to the ISA); in this new section, we will now consider architectures where the hardware plays more than a supporting role in code-size reduction. Actually, a variety of hardware choices all play "leading roles" in the final code size (for instance the choice of ISA, the number of addressable registers, and whether to use a VLIW or a superscalar). But many of these hardware choices are based on run-time or other considerations. Here, we will only consider hardware technologies that are specifically added for *decompressing* instructions.

One such category of hardware modifications is to provide multiple ISA formats. Both the ARM Thumb [19] and the MIPS16 [20] employ this strategy, defining both 16-bit and 32-bit formats of their instructions. A special instruction is also provided for switching between these two formats. Since the 16-bit format has reduced widths for its immediate and register fields, code compiled to the 16-bit form will typically require additional instructions and suffer a performance penalty. Yet, if infrequent regions are properly identified for this optimization, significant code reductions are possible. This approach suffers from the hardware complexity of maintaining two separate ISAs at the same time. While neither ARM nor MIPS are VLIWs, it is possible to implement a similar scheme within a VLIW. In this case, the benefits of our approach would be reduced because the NOPs within the long word could be expressed in the more-compact form.

A more general application of this hardware compression principle is proposed in [15]. Here, the compiled program is first analyzed to identify its optimal instruction width; for instance, a small program will not have large branch offsets. The entire program is then compiled for this instruction width. The compiler also generates an HDL description of the decoder that can convert these reduced instructions back to their 32-bit forms. When the program is run, first the customizable decoder is programmed based on the HDL description. Unlike the 16-bit versus 32-bit choice of [19] and [20], the approach of [15] allows for a variety of bit widths. A second difference is that the bit-width is fixed for the entire program. Being an extension of the more-simple hardware schemes discussed in the last paragraph, [15] compares to our method in much the same way as they do.

Going beyond customizing only the decoder logic, [16] proposes a method of code size reduction for custom embedded processors by choosing customized templates so as to minimize the NOPs for a given set of applications. Instead of being a compiler optimization, however, this work solves a different problem of design space exploration for customizing hardware.

Another hardware method used for compression is CodePack [49], which is used in the PowerPC. PowerPC instructions are 32-bits wide, but the compiler, using CodePack, partitions each instruction into two 16-bit halves. Then each half is separately compressed into a variable-length code-word. When the PowerPC fetches these compressed instructions, it consults its dictionaries and decodes the instructions *prior* to their placement in the I-cache. The advantage of storing decompressed instructions in the I-cache is that, since most instruction fetches result in successful hits to the I-cache, the common-case becomes easier, and decompression is not needed on every cycle. One disadvantage, however, is that branch addresses must be translated, consuming extra cycles.

Compared to our approach, the first distinction of CodePack is that it does not target a VLIW. The second distinction is that it does not target an embedded processor, but rather introduces new hardware for address translation that may not be desirable for many embedded applications. Therefore, even though CodePack is able to efficiently compress any NOPs that might occur in the code, it is not relevant to the types of machines that our approach targets.

2.4.1 Existing hardware approaches for VLIWs

The processors just described are superscalars; the code size of VLIWs is even greater, because there are not only instructions but also NOPs that must be placed into long-word slots, when there is either an insufficient amount of parallelism or a slot restriction that prevents its use by certain instruction types. Traditional VLIWs produce even more NOPs than this, because long-words filled with NOPs are also used to perform stall cycles; such stalls are needed when an input operand is not-yet-available by the time that the instruction containing the operand is ready to execute. Therefore, most modern VLIW architectures provide some form of hardware support to reduce code size. Since, as the next chapter will elaborate, we achieve our code-size reductions by placing instructions more compactly so as to increase the number of empty long-words, our method actually *depends* upon the hardware having a stall mechanism. Therefore, stall-cycle compression hardware is not a competing strategy, but rather a necessary condition for our methods. Fortunately, most modern VLIW do have some form of stall mechanism. In this section, we will overview five such methods.

First, in architectures that allow for *stall bits*, the number of cycles to stall is encoded into a special bit-field within the long-word. This approach avoids the use of NOP-filled long-words for stalls, by instead setting special bits. Such a method is described in [16, 17]. Our research is applicable, without alteration, to machines with stall bits.

Second, in *multi-NOP* machines, the NOP opcode contains an argument field that specifies the number of cycles to stall after executing the current long-word. Thus, by scheduling a single NOP in the last cycle before the stall, empty long-words are avoided in the code. An important example of such a processor is the Texas Instruments TMS320C6x [18]; an older one is [21]. Multi-NOP processors reduce code-size by removing some of the NOPs within the code, but many of the NOPs will still remain. Not only are NOPs used to perform stalls and to sometimes separate long-words, but they are also needed for handling slot restrictions and for aligning packets, such as in the TMS320C6200 and TMS320C6400. Our research is applicable to multi-NOP machines, because they still maintain NOPs (or multi-NOPs) in their binaries. The benefit of our approach will be limited, however, by the number of NOPs placed into one multi-NOP.

Third, in *EPIC* architectures, NOPs are avoided in two ways: 1) hardware detection of data hazards spares the compiler from generating stall cycles, and 2) *tem-*

plates allow for variable-width long-words. In EPICs, parallelism is not indicated by the width of the fetched instruction, but by special *stop-bits* that are encoded within the templates. All instructions between two successive stop-bits are considered to be parallel. Since multiple stop-bits may occur within a single fetched-instruction address, EPIC processors can express variable-width parallelism, thereby avoiding the need to pad long-words with NOPs to make them the width of the VLIW. Yet, EPICs, like multi-NOPs, still contain many NOPs. In EPICs, NOPs arise from several sources: instruction type restrictions within templates, lack of a template with a stop-bit at the desired position, and alignment of long-words to avoid cache-block boundary penalties. An example of an EPIC is the IA64 [13]. IA64 is not an embedded processor, but similar designs have been proposed for embedded systems, such as TEPIC[15]. Our approach is applicable to EPICs, because of the NOPs that arise from template restrictions.

Fourth, in *dictionary-based* processors, such as Infineon's Carmel Processor[22], individual long-words are stored in a hardware dictionary very similar to the one described in [33], and is previously discussed in Section 2.3. Instead of the call-dictionary instructions of [33], however, Carmel introduces CLIW (Custom Long Instruction Words) instructions. While the call-dictionary instructions contained an argument to specify the length of the code segment, CLIW instructions assume a fixed length equal to the width of its VLIW core (6 instructions). Moreover, the instructions in a dictionary entree must be parallel, since, unlike in [33], the Carmel processor is a VLIW. Also unlike call-dictionary instructions, the CLIW instructions contain four register fields. When a VLIW instruction is retrieved from the dictionary, these register fields are inserted into the operand fields of the instruction packed into the long-word. This method allows not only for a more compact dictionary entry (since only CLIW register argument positions are needed), but also for more flexibility for reusing the dictionary entrees in situations where the registers would be changed for different call sites. Using these CLIWs, Carmel is able to reduce the code size and run-time of frequent code. The proper scheduling of these CLIWs can reduce not only the code size of their declarations, but also contention in the small hardware table that stores the CLIWs. Other than these CLIW instructions, Carmel functions as a 2-wide VLIW, with a variable-width that allows for scalar (1-wide) operation. NOPs still arise in the Carmel processor for a variety of reasons, however. These reasons include: that the instructions creating the CLIW instructions must write NOPs into some dictionary entree slots (implicitly increasing both the code size and the contention inside of the small dictionary), that scalar NOPs are needed for stall cycles, and that NOPs are used for alignment of basic-block boundaries. Since these NOPs are present in the Carmel, our code-size reducing methods can be applied.

Fifth, in *VLES* (Variable Length Execution Set) architectures, NOP instructions are stored in an encoded form in the executable program [23]. This encoding may be achieved either through an explicit stop-bit (TigerSharc [24]), or through a prefix count that indicates the width of the current long-word (StarCore [25]), or through a special bit-mask header to indicate specific NOPS within the long-word, which simplify unpacking (Phillips Trimedia [26]). Since the NOPs do not appear directly in the binary, these machines have a variable-length-execution-set (VLES)

– VLES is specifically the term used for StarCore [25], but for lack of a better term, we will use this designation to refer to all such variable-length VLIWs. When fetching compressed instructions, the designers must decide whether to store them in the I-cache in compressed or in un-compressed form. If they are stored in compressed form (like Trimedia [26]) then decompression must occur prior to instruction decode, likely leading to an extra pipeline stage. In order to avoid lengthening the pipeline, the instructions could be decompressed in the I-cache [29], but then there will be the same issues with inconsistent instruction addresses as those that are described in Section 2.4, for the CodePack [49] compression method.

Most VLES systems do not stop with simply compressing NOPs, but also compress the length of other instructions; although each processor has its own approach to this problem, we will now present the compression mechanism of the popular Trimedia processor, as an example. Trimedia defines four separate instruction widths: 0 bits (for NOPs), 24 bits (for the most frequently-used instructions), 36 bits (for somewhat frequent instructions), and 42 bits (the uncompressed length, for infrequent instructions). In order for the decompressor hardware-unit to unpack these instructions, it must know which instructions have been encoded with each method. Therefore, each long-word is, conceptually, prefaced by a header containing 2 bits per instruction. In point of fact, however, the long-words do not really contain headers but rather footnotes, because, in order to speed up the decompressor unit, the header for any particular long-word is actually stored at the end of the previous long-word. This solution presents a problem for branch-in points, because their previous long-word is indeterminate. Therefore, a fixed-size header is assumed for branch-in points. In addition, each long-word is padded with wasted bits to make it byte aligned.

Considering the hardware described above, Trimedia's code compression has three basic costs. First, like any new hardware, the decompressor unit must occupy chip real-estate, thereby increasing the dollar cost and the power requirements. Such increases are of greater concern to embedded processors than for desktops, because embedded chips have significantly smaller dollar and power budgets. Second, there is a run-time cost that arises from the added pipeline stage. Since decompression occurs early in the pipeline, it requires an extra branch-delay cycle. Typically, compilers have difficulty filling even two branch-delay long-words; decompression results in a the third branch-delay long-word which may often not be filled. Unfilled long-words degrade performance because they, in effect, become stall cycles. Third, there is a code-size overhead to Trimedia's decompression strategy. Of course, decompression is implemented for the sole purpose of reducing code size; yet, the overheads inherent to the approach reduce the amount of this code-size reduction.

This code-size overhead results from four sources. First, every long-word has extra bits for its header. Second, every long-word must be byte-aligned, which wastes additional bits. Third, since there is no stall mechanism, a long-word full of NOPs must be used. Although such long words are compacted, they still must contain the header information. Fourth, when the branch-delay cycle associated with the decompression stage is not filled, this also introduces another empty long-word that would not have been needed were there not a pipeline decompression stage.

It is clear from the variety of research that compression is an important area of research, and also that there are hardware/software trade-offs. Generally speaking, the hardware solutions tend to have much higher compression ratios. Yet, including new hardware adds cost and power consumption. These costs are particularly undesirable in embedded systems, which are the very applications most concerned with code size. Therefore, compiler based compression is still an active area of research, despite the fact that hardware appear to be better.

The very fact that VLIWs are popular in embedded systems reveals that the concerns of embedded systems are different than those of desktops. VLIWs are known to have a lower performance than superscalars, but many embedded applications prefer saving cost and power, rather than achieving slightly faster execution. VLIWs are also known to achieve this reduced hardware complexity through increasing the complexity of the compiler; but again, many embedded systems are less-concerned with compile-time, since the end user never has to experience this delay. Therefore, before choosing to add new hardware and new pipeline stages to a VLIW processor, the designer must first consider the needs of his applications and the possibility of using compiler solutions. If the compiler solution yields sufficient improvements, then it should be preferred.

Sometimes, a variety of technologies can be used for additional benefits. This is certainly the case when considering our new compiler method in addition to existing compiler methods. But this is less certain when both compiler and hardware solutions are simultaneously used to attempt to reduce code size. After all, information theory assures us that there is a definite lower bound on compression, so that, as more optimizations are added, the amount of available benefit from the other technologies diminishes. Since hardware solutions yield large reductions, including additional compiler methods in such systems will almost always indicate smaller improvements for the compiler methods than they would achieved on their own. In addition, for our compiler optimization, only certain types of VLIWs are applicable. Therefore, the next chapter will consider in detail the applicability of our approach to the kinds of VLIWs just described in this current section.

Chapter 3

Applicable VLIWs

The specific VLIW architecture impacts the achievable benefit of our method in two ways. First, since we improve code size by reducing the number of NOPs in the code, architectures that tend to need more NOPs will provide us with more opportunities for reducing them. Second, since our within-trace algorithm works by separating computation from stall cycles, the underlying hardware's stall mechanism affects the applicability of our within-trace approach. In particular, older VLIWs that use full-length long-words filled with NOPs to specify stalls cannot benefit from our within-trace methods. A fuller description of hardware mechanisms for stalling and for NOP compression follows.

The only type of stall mechanism that does *not* benefit from our within-trace methods (but can still use our across-block methods) is when long-words full of NOPs are used to specify stalls. To understand this mechanism further, consider that the simplest VLIW hardware does not provide any mechanism for stalling. Therefore, the compiler must analyzing the data dependencies to identify where stall cycles are needed and then insert long-words filled with NOPs into the code at these points. In terms of code size, these NOP-filled long-words are an expensive way to achieve a stall. In such systems, there is little that our within-trace scheduler can do to reduce code size. For example, if a particular trace requires X cycles to execute, it will necessarily have a code size of X long-words and cannot be made smaller. Because of its high code size cost, modern VLIWs rarely use this stalling method.

In Section 2.4.1 a variety of VLIW processors are described, that all use some more advanced stalling mechanism; we will now briefly review these mechanisms. There are four stall mechanisms used in modern VLIWs: 1) hardware dependency checks, 2) stall bits, 3) multi-NOPs, and 4) using variable-width long-words for stalls. First, in machines with hardware dependency checks, the hardware detects when to stall by checking if any input operand in the long-word is not ready. Some examples are StarCore [25], TigerSharc [24], and IA64 [13]. IA64 is not an embedded processor, but similar designs have been proposed for embedded systems, such as TEPIC[15]. Second, in machines with stall bits[16, 17], the number of cycles to stall is encoded into special bits within every long-word. Third, in multi-NOP machines, such as the TMS320C6x [18], the NOP opcode contains an argument field that specifies the number of cycles to stall after executing the current long-word. Fourth, VLIWs such as Carmel [22] and Trimedia [26] may use their variable-width feature to achieve stalls at a reduced code-size cost. Although NOPs must be inserted into the code for every stall cycle, the variable-width feature reduces the impacts as compared to a simple VLIW that inserts a full long-word of NOPs. We note that not all VLIWs with variable-width long-words necessarily use this mechanism to achieve stalls, however. Most of the machines described above are also variable width-machines, but employ more advanced stalling procedures.

Our compiler method can apply in all four of the above classes of VLIWs, since they all avoid long-words filled with NOPs to specify stalls; at the same time, however, the amount of benefit that our method can achieve will vary significantly between these processors. The reason is that our method aims at reducing the NOPs in the code, and many of these architectures already contain hardware to accomplish the same purpose. The amount of our benefit is directly proportional to the amount of NOPs left in the code. Some other sources of NOPs affecting one or more of the above classes of machines are: alignment requirements (such as for the start of basic blocks or for long-words that cross cache-block boundaries), a lack of sufficient parallelism, instruction issue restrictions, stalls that require single NOP slots, and multi-NOPs. Our method reduces all of these sources of NOPs, in all four of these classes of VLIWs. Specific pruning strategies may need to be modified to target the unique features of some of these architectures, however.

3.1 An examination of the VLES hardware solution used by the Philips Trimedia processor

The impact of our methods is most limited for VLES-based processors (see Section2.4.1); some of these processors use encoding to completely eliminate NOPs within the long-words as they appear in memory. Therefore, it is interesting to compare the costs and benefits of these two approaches. To do so, we now describe a theoretical VLES architecture that is modeled after the Trimedia processor, one of the most successful VLES architectures. In order for comparison with our current methods, we will now describe a theoretical, 6-wide VLES machine that uses the

ISA and stall-mechanism of the Itanium, but the code-compression strategy of the Trimedia.

In Chapter 2, Section 2.4.1, we described the VLES hardware used in the Trimedia, and its costs; we now wish to consider the specific issues that arise for 6-wide machine based on this VLES mechanism. First, since each instruction in the Trimedia contains two extra bits that indicate its instruction type, a 6-wide implementation each long-word will therefore require an 12 extra bits of *header* information. Second, since uncompressed Trimedia instructions are 42 bits and IA-64 instructions are 41 bits, the code size of the instructions is comparable.

In addition to these extra bits for headers and alignments, the VLES approach to code compression results in two additional, more-subtle code-size *costs* that further serve to decrease the amount of code-size improvement that the technique achieves. The first of these costs arises from the fact that each instruction's header information is actually contained in the previous long word, as described in Chapter 2, Section 2.4.1. In such a scheme, control-flow introduces ambiguity, because a single header may map to two different next-instructions, and also because multiple headers may map to the same next-instruction. The Trimedia's solution to this ambiguity is to use a fixed-size header (rather than the header of the previous instruction) for all branch-in points. As a result, the first long-word after a branch-in point will not be compressed, and its NOPs will still need to be inserted into the memory. The second of these more-subtle code-size costs arises as a consequence of the decompression pipeline stage. We have already noted the run-time and chip-area costs of the decompression unit, in Chapter 2, Section 2.4.1; here we present its effect on code size. Since he Trimedia, like many VLIWs, employs branch-delay long-words to avoid stalls, every pipeline stage prior to the execution stage (where branch targets are resolved), will result in a branch-delay long-word; by introducing a decompression stage, VLES thereby requires an extra branch-delay long-word, in addition to the two already needed for fetch and decode. Branch delay slots are notoriously difficult to fill, and it is likely that introducing a *third* such long-word will not be fully utilized. Under-utilization leads to additional long-words, and therefore to extra header bits. We model this situation by including three branch-delay slots into our compiler.

One final difficulty in constructing our comparison machine is that the Itanium ISA was not designed with compression in view, unlike the Trimedia. On the Trimedia, many instructions are not the full 42 bits, but are actually compressed into 24 or 32 bits. When translating this behavior to the IA-64 ISA, the question arises: "which IA-64 instructions should be considered to be compressed to 24 or 32 bits?" Since there is no realistic way to answer this question, we instead choose to compute *lower and upper bounds*, assuming (in the one case) that all instructions are compressed to the smallest size (24 bits), and (in the other case), that no instructions are compressed.

A VLES with perfect instruction compression is modeled by simply assuming that all IA-64 instructions have a 24-bit equivalent form. Clearly, this will not be the case for most instructions; yet, since the Trimedia compression mechanism targets commonly-used opcodes, compression may be possible for the majority of actual instructions within long words. For a cycle containing a single useful instruction, perfect compression will produce a 40-bit long-word (24 bits for the instruction + 12 bits for the header + 4 bits for alignment).

Assuming perfect compression provides a true lower bound on the amount of code-size compression that could be achieved if our target ISA were carefully divided into compressable and non-compressable instructions. In so dividing the instructions, certain instructions may occur in reduced form, perhaps with narrower register fields; yet, since these ISA changes can never decrease the number of instructions needed, the lower-bound guarantee is not weakened.

A VLES which only compresses NOPs serves as the upper bound. We observe three facts about this upper bound: 1) this bound is likely to be extremely conservative, 2) since VLES is a competing compression technology to our methods, comparing against an upper-bound is less-useful than against a lower-bound, and 3) a VLES can be imagined that only compresses NOPs. In such a VLES, each instruction would only need a single bit of header information, to indicate whether the slot is a NOP or an instruction.

Although not typical of most VLES systems, an architecture that only compresses NOPs is an interesting comparison machine; it separates the code reduction achieved by removing NOPs from the additional reduction achieved by compressing instructions. Since our methods also deal only with NOPs, this theoretical machine allows us to compare the software and hardware approaches to NOP reduction. In this comparison, we note that, since instructions are not compressable on such a VLES, we do not need to introduce a lower-bound, as we did for the perfect-compression VLES, above. In a 6 wide VLIW employing such a NOP-only compression scheme, a long word containing a single useful instruction would therefore require 6 bits for the header, 41 bits for the one instruction, and one addition bit for alignment – a total of 48 bits.

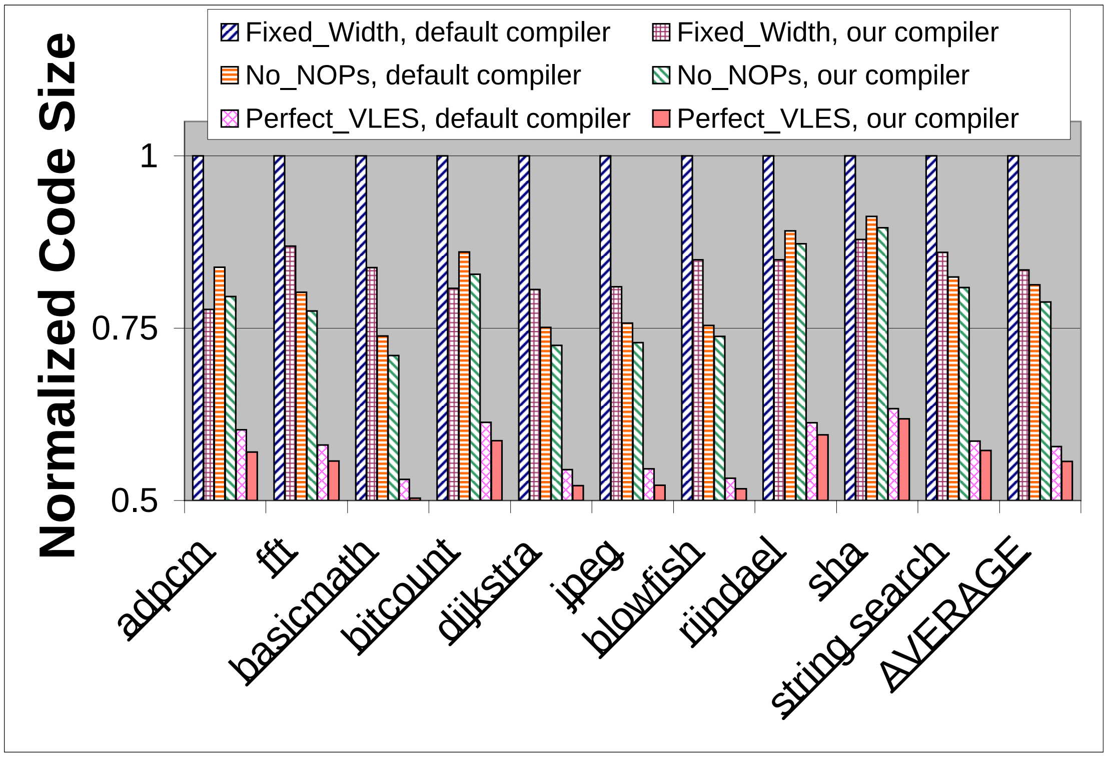


Figure 3.1: Normalized code size for three test machines, both with and without our compiler optimization.

Figure 3.1 presents the effect of our technique on three architectures, normalized to the code size of the fixed-width VLIW using the default compiler. The first of these architectures, labeled *Fixed* is the standard 6-wide fixed-width VLIW used in all of our results. The second machine,*No-NOPs*, represents the hypothetical, 6-wide NOP-compressing VLES architecture that uses a 6-bit header for each long-word. The third machine, *VLES*, is a 6-wide VLIW that assumes perfect instruction compression. As described above, each operation is compressed to 24 bits and each header is 12 bits. This machine serves as a *lower bound* for the achievable compression in a more-realistic Trimedia-like machine. For each of these three test-machines, we consider the code-size that results both from the default compiler, and from the methods of this paper.

Comparing the six bars of Figure 3.1 reveals several key insights The benefit of our compiler methods over the default machine is found by comparing the first and second bars (16.6%). Although the overall improvement of our scheduler is elsewhere in this paper reported as 16.3%, we here obtain a slighty hiher value because, since VLES machines typically allow predication, we have included the predication-based optimization of Chapter 9 into our method. Looking again at Figure 3.1, the benefits of including our methods into each of the hardware compression schemes is found by comparing the third bar with the fourth (a 2.5 % improvement for the NOP-compressing VLES) and by comparing the fifth bar with the sixth (a 1.5% improvement for the perfect compression VLES). Also note that, for the default compiler, the benefit of a NOP-compressing VLES scheme is found by comparing the first and third bars (19%), and the benefit of a Trimedia-like VLES scheme is *lower-bounded* by comparing the first and fifth bars (43%). Yet if our compiler is used, the extra benefit of a NOP-compressing VLES is 6.5% (by comparing the second and fourth bars), and the benefit of a Trimedia-like VLES scheme is some number *less than* 29.5% (the lower-bound found by comparing the second and sixth bars).

In considering these results, we make four observations. First, our NOP-reducing scheme (second bar) is nearly as effective as hardware-based NOP-compression (third bar). Second, while both hardware techniques enjoy larger reductions than our compiler-based approach, they also suffer from run-time, power and dollar costs that are not reflected in this code-size figure. In contrast, our compiler method has only a compile-time cost. Third, our compiler methods provide for only modest improvements on VLES machines. Fourth, hardware methods conversely also provide much smaller improvements when implemented into a system using our compiler-based methods. In fact, whenever two technologies are employed to tackle the same problem, they often yield a combined improvement better than either one used separately, and yet the *additional* benefit achieved by adding the second technology is usually smaller than the benefit that this technology achieves by itself. The designer must decide which of the technologies to use (or whether to use both) based on an analysis of the costs and benefits. While it is fair to ask whether the implementation of our optimization into the compiler would be worth the effort on VLES systems, it is also reasonable to ask whether the implementation of expensive hardware methods is worth the trouble for embedded systems, now that such a compiler method is available. Using the default compiler, the system designer may choose to implement VLES, anticipating a code-size improvement of up to a 43% (assuming perfect-compression). Since our method reduces this benefit to less than 29.5%, however, the designer may now make a different choice.

In fact, as described at the end of the last chapter, our methods can be used in conjunction with many of the existing, compiler-based code compression schemes, because our method targets a different source of reduction than do many of the previous methods. As a variety of such compiler strategies are employed, the code size begins to approach its theoretical limit, and the additional benefit obtainable from VLES will become even smaller. Ultimately, the system designer's choice of whether to include VLES hardware will depend on his target applications and system requirements.

In the case of hardware versus software solutions, there is an on-going debate. In most cases, the hardware solution is both more-obvious-to-implement and more-effective, but also has undesirable costs that the software solution does not have. Compiler research is particularly critical for embedded systems, because as the compiler technology catches up with the benefits achieved by hardware methods, the scale begins to tip in favor of compiler-based approaches. Indeed, the very choice of a VLIW processor indicates that the application environment calls for a simplified processor with extra compiler support.

3.1.1 A note on VLES systems that store decompressed long-words in the I-cache

Reviewing the above discussion, many of the costs of VLES are found to arise from the extra pipeline stage that is needed for decompression. To reduce the impact of decompression, it is possible to extend the VLES approach to decompress instructions prior to their insertion into the I-cache, as proposed in [54] (for Trimedia) and as previously employed by CodePack [49] (for a desktop superscalar). Decompressing instructions before placing them into the cache has the advantage of speeding up the common case; decompression will not be needed on I-cache hits. The disadvantage, however, is that branch addresses must be converted since the I-cache and the main memory are no-longer consistent. On such a VLES system, our approach becomes useful as a technique for shrinking the footprint of the code in the I-cache.

Chapter 4

Optimal base algorithm for scheduling instructions within traces for code size

We now describe our within-trace, back-tracking instruction scheduler. This chapter will begin the description by presenting a simple, exhaustive-search instruction scheduling algorithm that is provably optimal (in the sense that it will find a solution with minimum code size among those with minimum run-time). This algorithm is not feasible, however, since its compilation time grows exponentially with the size of the trace. Therefore, Chapter 5 will improve upon the algorithm by adding aggressive and novel branch-and-bound techniques that are used to prune portions of the search space while retaining the optimality guarantee. Although these additions drastically reduce the compilation time, the search space remains potentially exponential, so that these techniques, while sufficient for most traces, are not sufficient for some others. To tackle these remaining traces, Chapter 6 will describe non-optimal heuristics that are used to guide the search toward more promising solutions quickly.

Despite these optimizations, it remains true that a backtracking-based algorithm pays a price in compile-time versus one-pass methods. Nonetheless, we believe this is acceptable – not only because our optimizations allow the compile-time overhead to be modest and user adjustable – but also because the compile-time is less important in embedded systems than in desktops, since the end-user never has to wait for code to compile on an embedded system.

This current chapter, presenting the non-optimized, simple search algorithm, is divided into 3 sections. In Section 4.1, the base algorithm is presented. Then Section 4.2 discusses the complex implementation details of this algorithm, which centers around the design and usage of the Finite State Automata method of identifying valid long-words. Lastly, Section 4.3 modifies our FSA methods to account for the presence of instruction ordering restrictions within long-words.

4.1 Introducing the base algorithm

The base algorithm, shown in Figure 4.1, performs an exhaustive search that returns a provably-optimal schedule for each trace. An exhaustive search is impractical, but serves as a good basis for an algorithm that can be modified to be feasible. The search is recursive; at each recursive step, the **for** loop examines all possible *instruction groups*, IG, from among the set of ready-to-schedule instructions, R. An instruction group is any set of instructions that may be executed in parallel.

In Figure 4.1, every IG must contain all critical instructions [1], C, as well some subset, NC_{subset}, of the non-critical-but-ready instructions. For each chosen IG, the algorithm first tests whether the processor can schedule all of the instructions of

[1]An unscheduled instruction is said to be critical (or to have met its deadline) if, intuitively, delaying it further will mean that a minimum latency solution cannot be obtained on this path. The minimum latency is the latency of the longest latency path in the Data Flow Graph. Mathematically, the condition for an instruction to be critical is that the *current schedule cycle + longest path latency from this instruction to the bottom of the DFG = minimum # cycles required to schedule the trace.*

```
SCHEDULE_RECURSIVE(U, PrevIG, PrevR)          // U:the set of not-yet-scheduled
            // insts, PrevIG: insts scheduled on last cycle, Prev R: insts ready on last cycle
  ADVANCE_CLOCK(Clk)                          // Advance the clock
  define Best = MAXINT                        // Initially, no best solution
  if ($U = \emptyset$)                        // See if finished
    return 0
  define $R$ = READY($U$)                     // The set of ready-to-schedule insts
  define $NC$ = NOT_YET_CRITICAL($R$,Clk)     // All R that could be delayed
  define $C = R - NC$                         // All R that must schedule now
  for each $NC_{subset}$ combination of elements of $NC$
    $IG = C + NC_{subset}$           // this inst group: all critical + some non-critical
    (Ccost) = FITS_IN_1_LONG_WORD(IG)         // Finds cost of current selection
    Rcost= SCHEDULE_RECURSIVE(U-IG,IG,R)      // Cost of rest (U-IG)
    cost = Ccost + Rcost
    if (Best > cost)                          // Is this solution the best so far?
      Best = cost
  end for
  return Best                                 // All possibilities have been explored
end
```

Figure 4.1: Optimal base algorithm for instruction scheduling.

IG into one long-word – a non-trivial analysis that will be the focus of Section 4.2. For now, its sufficient to conceptually understand that only instruction groups that are schedulable in one long-word will be considered further. For each of these instruction groups, a recursive call finds the best schedule of the remaining code, and the scheduling clock advances. If an IG contains fewer instructions than the width of the VLIW machine, the remaining slots are padded with NOPs. In its present formulation, Figure 4.1 is for fixed width VLIWs, but the modifications are slight to accommodate systems with variable-length long-words, such as multi-NOP or EPIC machines.

In order to identify the critical instructions, we need to know how many cycles will be required to schedule the entire trace. Although this number cannot be easily calculated we can identify a lower-bound approximation that is often correct. This initial estimate is itself the maximum of two lower bounds. One of these lower bounds is the height of the DFG for the trace, because this height indicates the longest computation path. The other lower bound is found from resource usage, as follows. For any resource, R, we may define $I(R)$ to be the number of instructions within the current trace that require resource R, and $N(R)$ to be the hardware-based limit on the number of such instructions that may be scheduled in a given cycle. For instance, if the hardware contains two floating-point units, then $N(R)$=2, because only 2 floating-point instructions may be scheduled in a given cycle. With this terminology, we may now define the second lower bound on run-time cycles for scheduling a trace: $2nd\_Lower\_Bound = MAX(I(R) \div N(R)), for\ all\ R.$

Once the lower-bound-estimate of run-time is found, the algorithm in Figure 4.1 may be executed. If this algorithm finishes without finding a solution, then all instruction deadlines are increased by 1, and the algorithm is re-run for that trace. For most traces, the original estimate is correct. In this way, only schedules of minimum run-time are considered.

Cost metric The goal of our search technique is to find the schedule with the smallest code size among those that have the minimum run-time. Thus the first constraint for scheduling is run-time. Because instructions must schedule by their deadlines, *Figure 4.1 always finds a minimum run-time solution*, albeit with an unrealistic compile-time. Among these solutions, the code-size cost is measured in NOPs, because the number of NOPs used is a direct measure of code size (since the number of useful instructions remains fixed, so long as instruction scheduling is performed only after instruction selection is finished). After the search completes, the solution with the lowest code size among the ones with minimum run-time is returned.

Impact of register allocation Instruction scheduling and register allocation are interdependent. The register allocator maps variables onto machine registers. Since a function may contain more variables than there are physical registers, the allocator must reuse registers. To reduce the cost of register spills, the allocator attempts to assign variables in such a way as to minimize the number of registers used by a procedure. If instruction scheduling is performed prior to register allocation, then the live ranges of all variables become fixed, and the allocator will not be able to reduce the number of registers as much as it otherwise could. On the other hand, if register allocation is performed first, then different variables will be mapped to the same physical register, thereby introducing *anti* and *output* dependencies into the DFG. And adding edges into the DFG will restrict the instruction scheduler, worsening its result.

We have chosen to implement our scheduler *after* register allocation. This avoids four problems that arise when scheduling before allocation. First, our algorithm would need a new heuristic for reducing register pressure in the schedule. Second, our schedule could cause new register spills. Third, such register spills generally require the allocator to insert new "spill" instructions that might not fit into the existing schedule. Fourth, the implementation would be harder, requiring modifications to the register allocator. But scheduling instructions on the final pass avoids all these problems, and allows us to ignore register allocation, by treating anti and output dependencies just like true dependencies.

We do note, however, that by moving our pass prior to register allocation, we could remove edges from the DFG, thereby improving the scheduling flexibility, which is likely to further improve the results. This increased scheduling flexibility would come at the expense of additional register spills, because reduced aliasing will increase register pressure. When more registers are needed, then registers from the caller function must be saved to memory – an expensive operation. Therefore, our base algorithm executes after register allocation and does not attempt any register re-allocation, but rather, simply treats all dependencies as true dependencies. In Chapter 8, however, we will re-visit register allocation and extend our methods to some situations where it is beneficial.

4.2 Using Finite State Automata methods to schedule instruction groups

Figure 4.1 contains a call to a function named "FITS_IN_1_LONG_WORD(IG)"; this section describes that function. The analysis is based on the Finite State Automata (FSA) methods presented in [1]. In considering the "FITS_IN_1_LONG_-WORD(IG)" function, we first note that, in this section, the set IG is to be understood as being unordered. This means that the instructions that are chosen to be scheduled in parallel must be entirely independent of each other. (The next section will consider how to handle ordering restrictions within the IG set).

To determine whether a given set of independent instructions, IG, are able schedule on a single cycle, an FSA is used. An FSA describes a syntax for legal sentences in a grammar. As described in [1], our state machine is designed so that all legal schedules correspond to sentences with a proper syntax in the FSA. To determine that a sentence has legal syntax, one starts with an empty sentence and adds words until either the grammar is violated or the sentence ends. In our case, the words represent instructions. Starting with an empty schedule, instructions, I from IG are chosen in any order. As each instruction is selected, the next state of the FSA is found by examining the current state and following the edge corresponding to the type of the instruction, I. Sometimes the current state will not have an edge of the proper type for instruction I. In such cases, the instructions of IG cannot all schedule on one cycle, because the lack of an appropriate edge indicates an illegal sentence in the grammar. The benefit of the FSA approach of [1] is that it provides an efficient means of pre-computing instruction interactions, so that the scheduler does not need to keep track of the actual resources of the machine – this task has instead been abstracted down to simply walking a state machine.

As in [1], our FSA is entirely machine dependent. Even though the method is applicable to any architecture, the FSA must be constructed specifically for that architecture. Therefore, this section contains many details that are only relevant to our test machine – other than their relevance as an example for those who wish to construct such an FSA for a new system. *Although the information contained in this section (and the next) is novel and also necessary for completeness, it is not necessary for the reader seeking a conceptual-level understanding.* Such a reader may safely continue to the next chapter, since the subsequent chapters are not dependent upon the following information.

Itanium Details Since the FSA is machine specific, we must preface its discussion with an overview of the features our test processor. To begin with, our processor is based on the Itanium processor [60], but it is a 6-wide, fixed-size VLIW, not an EPIC [6]. Nonetheless, the hardware resources are that of an Itanium. The Itanium has two integer ALUs (I), two memory units (M), two floating point units (F), and two branch units (B). (There are no A or LX units, because A-type instructions use either an I or an M execution unit, and LX-type instructions use both an I and an F unit).

Itanium manages functional unit assignment through *templates*. An instruction's template is preserved in special bits within the long-word; the hardware uses these bits to determine the functional types of the operations within the long-word. Some operations are able to execute on different types of functional units, but the template informs the hardware of the intended assignment. The templates on an Itanium are each three instructions wide, but the parallel regions may span any number of templates, because *stop bits* indicate the parallel regions, as described in Section 2.4.1. Although fixed-width VLIWs are more common than EPICs in embedded systems, the available Itanium compiler is more mature than many that are publicly available for embedded processors. In addition to historical reasons, we therefore decided to use the EPIC compiler to target a fixed width VLIW. This is not difficult, since the EPIC representation is more general than the fixed-width one – we obtain a 6-wide VLIW by simply inserting stop bits after every 6 instructions.

We also need to convert Itanium's 3-instruction-wide templates into 6-instruction-wide templates for our test machine, in such a way that the new templates do not exceed the Itanium's functional units or other resources. For simplicity, we do not discuss this template conversion process here, but have instead described it in Appendix A, which contains many of the derivation details for this section. Appendix A also contains a full description of templates.

Performing the calculations of Appendix A results in Figure 4.2, which shows that our 6-wide, fixed-width VLIW has 27 possible templates; all other possible type-orderings being disallowed by the Itanium's resource limitations. Template restrictions have a real effect in EPICs, because the internal-stop-bit templates offer few choices; for our fixed-width VLIW however, only a few templates are disallowed on these grounds. Many more template were disallowed because they exceed the resources of the Itanium.

M	I	I	M	B	B
M	I	I	B	B	B
M	I	I	M	F	B
M	LX		M	LX	
M	LX		M	F	I
M	LX		M	I	B
M	LX		M	B	B
M	LX		B	B	B
M	LX		M	F	B

M	M	I	B	B	B
M	F	I	M	LX	
M	F	I	M	F	I
M	F	I	M	I	B
M	F	I	M	B	B
M	F	I	B	B	B
M	F	I	M	F	B
M	I	B	M	LX	
M	I	B	M	F	I

M	I	B	M	I	B
M	I	B	M	B	B
M	I	B	M	F	B
M	F	B	M	I	I
M	F	B	M	LX	
M	F	B	M	F	I
M	F	B	M	I	B
M	F	B	M	B	B
M	F	B	M	F	B

Figure 4.2: The 27 templates available for our 6-wide machine. (Derivation found in Appendix A.)

These 27 templates identify all legal instruction-type combinations, but the assigning of instructions to these slots is further complicated by the fact that not all hardware units of a given type are identical. Itanium has two copies of each of the I, M, and F units, yet these two copies are not exactly the same. For instance, one of Itanium's I-type execution units is *general* and the other is *restricted*, because it can only execute a subset of the I-type instructions. By implication, there are therefore certain *specialized* I-type instructions, that can only be executed on the general I unit. Thus, the meaning of a general execution unit and a general instruction are reversed from each other: the restricted I-type unit can only execute general I-type instructions, while specialized I-type instructions require the general I-type execution unit. In the following presentation, a lower case "i" may be used to denote either a specialized I-type instruction or a restricted I-type execution unit, while an upper case "I" denotes both general I-type instructions and the general I-type execution units. Therefore an "i" instruction may only use the "I" execution unit, where as an "I" instruction may use either of the "i" or "I" execution units. Similar nomenclature is employed for the M-type and F-type instructions and execution units. In addition, since some A-type instructions are also specialized, we further define an "a" instruction to be one that is executable only on an "I" or "M" unit, and an "A" instruction to be one that is executable on either an "i", "I", "m", or "M" unit. But we do not define A-type execution units, because these do not physically exist on the Itanium.

The templates in Figure 4.2 do not make any distinction between general and restricted execution units, because the Itanium applies a left-to-right assignment rule. For instance, we can see from the figure that most templates allow for two M-type instructions. The Itanium will assign the M-type instruction that occurs earlier (*i.e.,* more to the left in the long-word) to the general M-unit, and the later M-type instruction to the restricted m-unit. Some exceptions are described Figure A.6 of Appendix A, but this information is not needed for a conceptual understanding.

With this information about Itanium's assignment of instructions from templates to execution units, we may now proceed to describing the state machines used for our 6-wide VLIW, following the approach of [1]. One of the key contributions of [1] is the observation that partitioning the FSA can substantially reduce the number of states. This partitioning is accomplished along disjoint functional units. In our case the M and I units are not disjoint, because there are A-type instructions that may schedule on either of these units. All other execution units are disjoint, but we found that a single partition was sufficient to reduce the state tables to an efficient size. The M and I execution units form one FSA and all other units form the second FSA.

In Table 4.1, the FSA for the M and I units is considered. Because the A-type instruction may execute on either an M or an I execution unit, the A and I units must belong to a single Finite State Machine (FSA). Table 4.1, requires some explanation. In this table, the 30 rows correspond to the 30 states in the FSA. Each of the 30 states corresponds to a specific set of instructions resources that are already being consumed by the currently scheduled instructions. Looking at the first column of this table, each state is given both a state number and a set of consumed resources. For example, the 29th row contains "29:(111000)" in the first column.

Current State	Next State					
Case#:(iImMaA)	i	I	m	M	a	A
0:(000000)	6	5	4	3	2	1
1:(000001)	17	13	11	9	8	7
2:(000010)	19	14	19	10	19	8
3:(000100)	18	15	12	12	10	9
4:(001000)	19	16	-	12	19	11
5:(010000)	20	20	16	15	14	13
6:(100000)	-	20	19	18	19	17
7:(000002)	27	21	23	21	21	21
8:(000011)	25	21	25	21	25	21
9:(000101)	24	21	23	23	21	21
10:(000110)	26	21	26	26	26	21
11:(001001)	25	22	-	23	25	23
12:(001100)	26	23	-	-	26	23
13:(010001)	27	27	22	21	21	21
14:(010010)	28	28	28	21	28	21
15:(010100)	27	27	23	23	21	21
16:(011000)	28	28	-	23	28	22
17:(100001)	-	27	25	24	25	27
18:(100100)	-	27	26	26	26	24
19:(101000)	-	28	-	26	-	25
20:(110000)	-	-	28	27	28	27
21:(010110)	29	29	29	29	29	29
22:(011001)	29	29	-	29	29	29
23:(011100)	29	29	-	-	29	29
24:(100101)	-	29	29	29	29	29
25:(101001)	-	29	-	29	-	29
26:(101100)	-	29	-	-	-	29
27:(110100)	-	-	29	29	29	29
28:(111000)	-	-	-	29	-	29
29:(111100)	-	-	-	-	-	-

Table 4.1: FSA States for the M and I units of our 6-wide VLIW. (Derivation found in Appendix A.)

The meaning of the parenthetical 6-digit number summarizes the resources used: the first digit indicates the number of “i” instructions, the second digit indicates the number of “I” instructions, the third digit indicates the number of “m” instructions, the fourth digit indicates the number of “M” instructions, the fifth digit indicates the number of “a” instructions, and the sixth digit indicates the number of “A” instructions. The heading of the first column succinctly summarizes this encoding information by the expression “#:(iImMaA)”. Therefore, state #29 corresponds to one restricted I-type, one unrestricted I-type, and one restricted M-type (i=1, I=1, m=1, M=0, a=0, A=0).

The next six columns in Table 4.1, describe the edges of the FSA. If, from state #29, an M-type or an A-type instruction is added, we transition to state #30. All other instruction types are marked with dashes, because these instruction types obviously cannot be scheduled on the same cycle as those already included. For a particular row in Table 4.1, the summation of all of the parenthetical digits for that state will tell us how many instructions have been scheduled. For instance, state #29 has three instructions scheduled (1+1+1+0+0+0 = 3). In this way, we can see that no state has more than 4 instructions scheduled (since the hardware has only four M and I execution units). In addition, since each next state is derived by adding a single instruction type to the current state, we can also see that the next state always has a parenthetical-digit-sum that is larger by 1 than that for the current state.

Although there are no actual “A units” on our test machine, the A-type instructions must be maintained as part of the state, because the decision of whether to assign to a M or I unit may depend on later instructions. For instance, suppose that, from state #21 (i=1, I=0, m=0, M=0, a=0, A=1), an I-type instruction is added. Then, it would seem logical that the next state would be (i=1, I=1, m=0, M=0, a=0, A=1). But in fact, such a state does not exists in Table 4.1, and the table instead indicates that the next state should be #28 (i=1, I=1, m=0, M=1, a=0, A=0). This happens because the fact that both I-type execution units are already filled tells us that the A-type instruction must become an M-type instruction. In essence, the state that would have corresponded to "(i=1, I=1, m=0, M=0, a=0, A=1)" has been merged into state #28.

This type of state merging is critical to achieving a small table. As new instructions are added to the current state, prior instructions may move into more restrictive types. In the last paragraph, an A-type instruction became assigned to an M execution unit. Similarly, an M-type instruction may move to the restricted-M execution unit. For instance, from state #6 (i=0, I=0, m=0, M=1, a=0, A=0), a new M type instruction does not produce (i=0, I=0, m=0, M=2, a=0, A=0), but instead produces state #11 (i=0, I=0, m=1, M=1, a=0, A=0). Since two M-type instructions are needed, then one of them must be assigned to the more unrestricted execution unit, even though neither of these instructions specifically requires this unit.

Because this state table has only 30 rows, efficient data structures are possible. For instance, since 5 bits are needed to express a state number, and since there are 6 next states from the current state, the next-state transition edges will require a total of 30 bits, which fits within a single long integer.

Current State	Next State				
Case#:(bBLfF)	brp	B	L	f	F
0:(00000)	5	4	3	2	1
1:(00001)	14	10	7	6	6
2:(00010)	15	11	8	-	6
3:(00100)	16	12	9	8	7
4:(01000)	17	13	12	11	10
5:(10000)	17	17	16	15	14
6:(00011)	22	18	-	-	-
7:(00101)	23	19	-	-	-
8:(00110)	24	-	-	-	-
9:(00200)	-	-	-	-	-
10:(01001)	25	20	19	18	18
11:(01010)	26	20	-	-	18
12:(01100)	27	21	-	-	19
13:(02000)	28	28	21	20	20
14:(10001)	25	25	23	22	22
15:(10010)	26	26	24	-	22
16:(10100)	27	27	-	24	23
17:(11000)	28	28	27	26	25
18:(01011)	29	-	-	-	-
19:(01101)	20	20	-	-	-
20:(02010)	30	30	-	-	-
21:(02100)	31	31	-	-	-
22:(10011)	29	29	-	-	-
23:(10101)	-	-	-	-	-
24:(10110)	-	-	-	-	-
25:(11001)	30	30	-	29	29
26:(11010)	30	30	-	-	29
27:(11100)	31	31	-	-	-
28:(12000)	-	-	31	30	30
29:(11011)	-	-	-	-	-
30:(12010)	-	-	-	-	-
31:(12100)	-	-	-	-	-

Table 4.2: FSA States for the F, LX and B units of our 6-wide VLIW. (Derivation found in Appendix A.)

In Table 4.2, the FSA for the F, LX, B, and brp instructions is shown. The format of this table is analogous to Table 4.1, except that there are only four next-state transitions per row. The brp instructions are separated from the general B instructions because the Itanium allows the brp instruction to be scheduled in some instances where other instructions cannot be. (For an explanation see footnote "d" on Figure A.6, in Appendix A.) Because the F and B units are disjoint, this FSA could be partitioned further. Since the the state table already only contains 32 rows, however, it would be counter-productive to divide it further, because of the overhead of maintaining correctness across multiple FSAs.

Template limitations of the IA-64 test machine have reduced the number of states in Table 4.2. For instance, state #8 in this table indicates two "LX" instructions. We notice that there is no next state for adding a "B" type instruction, even though all of the B units are idle. The B-type instruction cannot be added, because there is no template in Figure 4.2 that contains two "LX" instructions along with a "B" instruction. Template restrictions were not a significant difficulty in the FSA of Table 4.1, however, because the IA-64 provides more templates for the common M and I instructions than for other units. [2]

Our machine therefore requires 62 states (30 + 32 = 62). This is a far smaller number of states than the machine studied in [1] because there are no inter-cycle restrictions in our machine. Every resource is fully pipelined so that the scheduler does not need to keep track of previous cycles as a part of the state – instructions

[2]Examining Figure 4.2, we see that 23 templates access both M units, and 16 templates use both I units. In contrast: 6 templates use all three B units, 4 templates use both F units, and 1 templates uses both LX units.

that are schedulable on a given cycle are not dependent on what had been scheduled on the previous cycle. But it must be remembered that a lack of inter-cycle *resource* dependencies is not to to say that there are no inter-cycle *data* dependencies. Data dependencies are handled in the algorithm of Figure 4.1 through the defining of the ready set, R; they are not the concern of the FSA, since it does not consider specific instructions, but only instruction types.

Collating states from two tables With two state tables, synchronization is required to maintain correctness. This issue arises because some state combinations exceed other overall-resource-constraints, beyond those considered in the individual tables. For instance, in Table 4.1, state #30 is a valid state that corresponds to using two M units and two I units. Similarly, the state #30 in Table 4.2 is also valid, corresponding to the use of all three B units. Though these two states are individually legal, yet they are mutually exclusive – it is not possible to use two M units, two I units, and three B units all at the same time, since this would require a 7-wide VLIW. Therefore, the VLIW width is one of the resource constraints that is not considered in either of these state tables.

The determination of whether two *half-states* (one from Table 4.1 and one from Table 4.2) are mutually compatible is accomplish through bit-vectors. For a given state in either table, there may be a variety of possible templates that can be used to schedule that state. For example, state #12 in Table 4.1 (which corresponds to two I-type instructions) may use any template from Figure 4.2 other than "MIIBBB", "MLXBBB", and "MFIBBB". By enumerating the 27 templates of Figure 4.2 as the numbers "1" through "27", we can express the set of templates that might schedule a given state as a bit vector; for state #12 in Table 4.1 the bit-vector is: "101111101111110111111111111" (little endian). Extending this same approach for all states from Table 4.1, we arrive at the bit-vectors presented in Table 4.3. In the same way, the states from Table 4.2 produce the bit-vectors that are shown in Table 4.4.

Testing for mutual compatibility between two half-states is accomplished through performing a bit-wise AND operation on the bit-vectors corresponding to each of the half-states. The AND-ing of these bit-vectors produces the set of templates that can schedule both of the half-states. Therefore, if this bit-wise AND produces an empty bit-vector, then the half-states are mutually exclusive. Such an approach for testing compatibility between half states is highly efficient due to three reasons: 1) each bit-vector can fit within one long-integer (27 templates $<$ 32 bits), 2) the bit-wise AND and zero-test operations each require a single CPU cycle, and 3) all of these state tables and bit-vector tables are pre-computed and built into the compiler as constants.

Putting it all together, Figure 4.3 presents the "FITS_IN_1_LONG_WORD(IG)" function that is called from within the algorithm of Figure 4.1. This function uses the FSA-based methods of the current section, to determine if a candidate instruction group is schedulable on one cycle. First, it initializes both FSA half-states. Next it begins including the instructions from IG, one at a time, in any order. For each instruction, its $Type$ is found, and the proper half-state is updated. If the update was unsuccessful, the instructions do not fit in one cycle. If, however, all

iImMaA	Template Number																										
Case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
5	X	X	X		X	X				X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		
6	X	X	X			X				X	X	X	X	X	X	X	X	X	X	X	X	X					
7	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
8	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
9	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
10	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
11	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
12	X		X	X	X	X	X		X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
13	X	X	X		X	X				X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		
14	X	X	X		X	X				X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		
15	X	X	X		X	X				X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		
16	X	X	X		X	X				X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		
17	X	X	X			X				X	X	X	X	X	X	X	X	X	X	X	X	X					
18	X	X	X			X				X	X	X	X	X	X	X	X	X	X	X	X	X					
19	X	X	X			X				X	X	X	X	X	X	X	X	X	X	X	X	X					
20	X	X	X									X	X					X	X			X					
21	X	X	X		X	X				X	X	X	X	X		X	X	X	X	X	X	X		X	X		
22	X	X	X		X	X				X	X	X	X	X		X	X	X	X	X	X	X		X	X		
23	X		X		X	X				X	X	X	X	X		X	X	X	X	X	X	X		X	X		
24	X	X	X			X				X	X	X	X	X		X	X	X	X	X	X	X					
25	X	X	X			X				X	X	X	X	X		X	X	X	X	X	X	X					
26	X		X			X				X	X	X	X	X		X	X	X	X	X	X	X					
27	X	X	X									X	X					X	X			X					
28	X	X	X									X	X					X	X			X					
29	X		X									X	X					X	X			X					

Table 4.3: Available Templates for the States of Table 4.1. Each row is one of the 30 states of that table, and each column is one of the 27 templates of Figure 4.2. An 'X' indicates that the corresponding template can accommodate the types in the current state. Each row forms a bit-vector for the corresponding state.

bBLfF	Template Number																										
Case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
1			X		X				X		X	X	X	X	X	X		X			X	X	X	X	X	X	X
2											X	X	X	X	X	X						X	X	X	X	X	X
3				X	X	X	X	X	X		X						X						X				
4	X	X	X			X	X	X	X	X			X	X	X	X	X		X	X	X				X	X	X
5	X	X	X			X	X	X	X	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
6												X				X								X			X
7					X				X		X												X				
8											X												X				
9				X																							
10			X						X				X	X	X	X					X				X	X	X
11													X	X	X	X									X	X	X
12						X	X	X	X								X										
13	X	X					X	X		X				X	X					X						X	
14			X						X				X	X	X	X		X			X	X	X	X	X	X	X
15													X	X	X	X						X	X	X	X	X	X
16						X	X	X	X								X						X				
17	X	X					X	X		X				X	X				X	X	X				X	X	X
18																X											X
19									X																		
20														X	X											X	
21							X	X																			
22																X								X			X
23									X														X				
24																							X				
25														X	X						X				X	X	X
26														X	X										X	X	X
27							X	X																			
28		X						X		X					X					X						X	
29																											X
30															X											X	
31								X																			

Table 4.4: Available Templates for the States of Table 4.2. As with Table 4.3, each of row forms a bit-vector for one of the 32 states in Table 4.2, and an 'X' indicates that the corresponding template can accommodate the corresponding state.

```
BOOL FITS_IN_1_LONG_WORD(IG)   // This function called by the algorithm of Figure 4.1.
// Input is an instruction group. Output is whether that instruction group is schedulable
    define MIA_State = 0                            // Initialize half of FSA State
    define FBL_State = 0                            // Initialize other half of FSA State
    for each I in IG
        define Type = INSTRUCTION_TYPE(I)   // Types are: M, m, I, i, A, a, F, f, B, b, LX
        if (ITS_AN_I_M_OR_A(Type))                  // See if the instruction is a M, I, or A
            MIA_State = NEXT_STATE(MIA_State, Type)   // Advance MIA half-state
            if (MIA_State = 0)          // Reverting to initial state means proper edge not found
                return FALSE
        else if (ITS_A_F_B_OR_LX(Type))            // If not M, I, or A, then its F, B, or LX
            FBL_State = NEXT_STATE(FBL_State, Type)   // Advance FBL half-state
            if (FBL_State = 0)          // Reverting to initial state means proper edge not found
                return FALSE
    end for

    // Individually, both half-states are valid. Now to test for mutual compatibility:
    define Possible_Templates = BIT_VECTOR(MIA_State) & BIT_VECTOR(FBL_State)
    if (Possible_Templates = 0)   // Mutually incompatible, no template works for both
        return FALSE
    return TRUE                                      // No conflicts, so it is schedulable
end
```

Figure 4.3: The FSA-based algorithm for quickly determining if a candidate instruction group is schedulable on one long-word.

instructions fishing being added without any errors, then the **for** loop ends and the mutual compatibility test is performed. If all of these test succeed, the instruction group IG is schedulable, and a value of TRUE is returned.

In the results, we also consider a 3-wide VLIW. The derivation of the 3-wide's unified FSA is also presented in Appendix A. In this case, the FSA is small enough to not require splitting.

4.3 Insuring correctness in the presence of intra-cycle dependencies

So far, this chapter has assumed that the processor does not allow any ordering restrictions within a long-word; we now consider the case when such restrictions are present. Although the instructions in VLIW long-words are theoretically independent, most VLIW processors actually allow some ordering restrictions, in the interest of improving the run-time. For example, consider the problem of dynamic memory disambiguation. Suppose that a memory-writing operation, M1, and a later memory-reading operation, M2, both have compile-time-unknown addresses, so that there can be no guarantee that M1 and M2 never refer to the same address. Then the compiler must schedule them in the original code-order. Without ordering restrictions inside of long-words, the compiler ensures this proper ordering by placing a one-cycle dependency edge between them. In a processor with ordering restrictions, however, these instructions can schedule on the same cycle, because the value being written by M1 may be *forwarded* to M2 whenever their addresses match. On such processors, the compiler will place a *zero-cycle* dependency between edge M1 and M2. Therefore, M1 may be scheduled either on an earlier cycle than M2, or on the same cycle as M2 – but it cannot be scheduled later. We refer to any zero-cycle dependency edges that occur within an instruction group, IG, as *intra-cycle* dependencies. We will also refer to the two instructions at the head and the tail of a given intra-cycle dependency edge, as its *intra-cycle instruction pair.*

Intra-cycle dependencies present a challenge to FSA-based instruction schedulers because the FSA only considers resource constraints. The FSA approach does not actually schedule the instructions within a long word; instead, it only indicates whether such a schedule exists, based on the resource constraints. For example, our FSA algorithm, shown in Figure 4.3, returns only a Boolean value. This is, in fact the very reason why FSA methods are popular – they reduce the compiler's task from that of placing instructions into specific slots to that of simply keeping track of resources. As a result, however, the FSA states do not have any mechanism to consider instruction ordering.

One intuitive-but-ineffective approach to allowing intra-cycle dependencies is to attempt all possible orderings of an instruction group, looking for any that both 1) map to a legal template, and 2) do not violate intra-cycle dependencies. In a 6-wide VLIW, there are 720 (= 5!) possible orderings. When we used this method, we experienced very high compile-time overheads. In addition, it defeats much of the purpose of constructing the FSA to begin with, as a schedule still ends up being generated.

Instead, we have devised a means of first simplifying the problem by dividing intra-cycle dependencies into three categories, and then solving each one separately. The three categories are: 1) intra-cycle dependencies that are guaranteed to prevent scheduling of the intra-cycle instruction pair on one cycle, 2) intra-cycle dependencies that are guaranteed to not affect the schedulability of IG, and 3) intra-cycle dependencies for which no guarantees can be made. Category 3 is the most difficult to solve, but fortunately most intra-cycle dependencies belong to either categories 1 or 2.

Category 1 dependencies occur when none of the 27 available templates can accommodate the intra-cycle instruction pair in their proper order – *even if this pair of instructions are the only ones in IG*. In our 6-wide VLIW, there are 4 possible ways to have a Category 1 dependency. First, an intra-cycle dependency whose tail instruction is a specialized m-type is guaranteed to be unschedulable. Such an intra-cycle dependency pair cannot be scheduled, because of two facts: 1) all 27 templates of Figure 4.2 begin with an M-type instruction slot, and 2) the left-most instance of an instruction type within a template maps to the general execution unit. Putting these two facts together, we see that a specialized m-type instruction must go into the first template slot; but if it goes into the first slot, then nothing can come before it, and so it *cannot* be the tail instruction of an intra-cycle dependency. Second, an intra-cycle dependency whose head instruction is an I-type (general or specialized), then the instruction is a specialized i-type also cannot be scheduled. This case arises because, in preserving the instruction order, the head instruction will go to the general I-unit, even though the tail instruction is specialized and also needs to use this same unit. Third, if the tail instruction is a specialized f-type and the head instruction is anything other than an M or an A-type (general or specialized), then the instruction pair is unschedulable because of two factors: 1) in Itanium, any F-type instruction that needs the general F-unit must be scheduled in one of the first three slots, and 2) in all templates that have an F-type in one of the first three slots, the F-type is always in the second slot. Then, since the first slot is always an M-type so that only a M or A-type instruction can go to the first slot, it follows that only these types may precede a specialized f-type instruction. Fourth, B-type instructions other than "brp" cannot be the head instruction unless the tail instruction is also a B-type. There are two reasons for this restriction as well: 1) among all B-type instructions, Itanium only allows "brp" and "nop.b" instructions to use the third template slot, and 2) other than the third slot, no B-type slot in any template is followed by a non-B-type slot.

Category 2 dependencies occur when every template that can accommodate the intra-cycle instruction pair can do so regardless of the instruction ordering. There are three cases. First, the pair is schedulable if both instructions are of the same type (other than A-type), and the tail instruction is a general type. In this case, the second instruction is certain to get the restricted unit, which is legal. Second, if the head instruction is an m-unit, then it will surely be assigned to the first slot, which is always legal. Third, if the tail instruction is a B-type (other than "brp"), then it is also guaranteed to present no problems because all branches occur at the end portions of the templates.

Category 3 dependencies are harder to detect, because they always involve three or more instructions. Table 4.5 presents the 8 cases that we identified. In this table, cases 1-3 all involve multiple intra-cycle dependencies. Looking at case 1, it prevents schedules that contain two M-type instructions when neither one can be the first instruction; such a case is clearly unschedulable, because there are only two M-units, and one of them always goes to the first slot. Also notice in case 1 that the head instructions cannot be M-types (because then there would be three M's and the FSA will already prevent it), and that the tail instruction cannot be a m-type (because this would be category 1 scenario). It is left to the reader to reason the remaining 7 cases. But we do note that some of the cases involve chains of three instructions, and require additional types to be present (although not necessarily part of a intra-cycle dependency).

Recall that, during the execution of Figure 4.1, Figure 4.3 is called, but that this FSA method does not consider intra-cycle dependencies; we now apply the

Case #	Possible Types of First Instruction	Possible Types of Second Instruction	Possible Types of Third Instruction	Other Types That Must be Present
1	AaIiFfBbL	$\longrightarrow$ M		
	AaIiFfBbL	$\longrightarrow$ M		
2	Ii	$\longrightarrow$ MIAa		
		$\longrightarrow$ MIAa		
		$\longrightarrow$ MIAa		
3	BbFfL	$\longrightarrow$ MIiAa		
		$\longrightarrow$ MIiAa		
		$\longrightarrow$ MIiAa		
		$\longrightarrow$ MIiAa		
4	Ff	$\longrightarrow$ F	$\longrightarrow$ M	
5	IiAa	$\longrightarrow$ MIiAa	$\longrightarrow$ ia	with: Mm
6	MmIiAa	$\longrightarrow$ IiAa	$\longrightarrow$ ia	with: Mm
7	AaFfBbL	$\longrightarrow$ ia		with: Mm, M
8	F	$\longrightarrow$ M		with: f

Table 4.5: Category 3 Intra-cycle dependency cases that are unschedulable.

observations of this section to consider how this technique might be extended for intra-cycle dependencies. First, adapting our scheduler to accommodate category 1 dependencies only requires converting the zero-cycle dependency edges into one-cycle edges. Not only does this change not lessen optimality guarantees, it is actually beneficially, since it prunes the search space of impossible combinations. Second, category 2 dependencies never have any effect on the schedule, so nothing needs to be done to the compiler to accommodate them. Third, category 3 dependencies present problems. Yet, since this case turn out to be somewhat rare, we can simply define a small list of various sets of instructions that cannot execute together. In most cases, this set is empty, and there is no compile-time overhead. If there are potentially-hazardous intra-cycle dependencies, however, then the current IG is compared against the illegal sets, using efficient bit-vector comparisons.

Chapter 5

Branch and bound to reduce compile-time

This chapter considers optimal pruning techniques to reduce the compile time of the scheduler described in Chapter 4. The next chapter will then explore non-optimal methods. In this chapter, Section 5.1 describes our pruning techniques for eliminating inferior schedules, and Section 5.2 explores methods of locating optimal solutions quickly.

5.1 Optimal pruning techniques

Since the run-time of the exhaustive search is infeasible, we use branch-and-bound techniques. Branch-and-bound techniques can drastically reduce the execution time while retaining the optimality guarantee, by *pruning* (*i.e.,* skipping) parts of the search space that are known to not contain an optimal solution, or if the remaining search space is guaranteed to have an optimal solution. Figure 4.1 contains a call to the *PRUNE()* procedure. If *PRUNE()* returns TRUE then the current path is abandoned by skipping to the next iteration of the **for** loop; otherwise it is explored further by making a recursive call.

Figure 5.2 shows the five pruning strategies we use in procedure *PRUNE()*. Each of these proposed methods has a low computational overhead; since these pruning strategies are executed at every step of the recursive schedule, high-overhead

```
SCHEDULE_RECURSIVE(U, PrevIG, PrevR)          // U:the set of not-yet-scheduled
            // insts, PrevIG: insts scheduled on last cycle, Prev R: insts ready on last cycle

  ADVANCE_CLOCK(Clk)                               // Advance the clock
  define Best = MAXINT                             // Initially, no best solution
  if ($U = \emptyset$)                             // See if finished
    return 0
  define R = READY(U)                              // The set of ready-to-schedule insts
  define NC = NOT_YET_CRITICAL(R,Clk)              // All R that could be delayed
  define $C = R - NC$                              // All R that must schedule now
  for each $NC_{subset}$ combination of elements of NC
    $IG = C + NC_{subset}$               // this inst group: all critical + some non-critical
    (Ccost) = FITS_IN_1_LONG_WORD(IG)              // Finds cost of current selection
    if ( not PRUNE(Ccost,U,R-IG,IG,PrevIG,PrevR))  // Skip unreasonable ones
      Rcost= SCHEDULE_RECURSIVE(U-IG,IG,R)         // Cost of rest (U-IG)
      cost = Ccost + Rcost
      if (Best > cost)                             // Is this solution the best so far?
        Best = cost
      if (Best = Ucost)                            // Is this a minimum-cost solution?
        return Best
    end if
  end for
  return Best                                      // All possibilities have been explored
end
```

Figure 5.1: Optimal algorithm with pruning methods added.

methods must be avoided. These pruning methods are general techniques applicable to any of the VLIWs our approach targets; other machine-specific pruning strategies can also be imagined.

This paragraph describes the first of our general pruning strategies, *COST_-PRUNING*. COST_PRUNING is applied both to the run-time cost *and* the code size cost. In either case, if the lowest possible cost of the current schedule is greater than or equal to the best solution found so far, then this path is deemed hopeless and is pruned. The lowest possible cost for the current schedule is computed as the sum of the cost of scheduling up to the current point plus a lower bound on the cost of the schedule the remaining instructions. The lower bound for the run-time cost is computed from the DAG height of the instructions that remain unscheduled,

```
PRUNE(Ccost, U, L, IG, PreviousIG, NewR)    // Inputs: current cost, the set of unscheduled
// instructions, the set of ready-but-not-chosen instructions, the chosen instructions, the inst-
// ructions chosen on the previous cycle, & the set of instructions that became ready this cycle
   define PartialCost = Cost of scheduling just those already chosen (everything but $IG \cup U$)

// COST_PRUNING: rejects solutions which are more expensive than the current best
   if (PartialCost+Ccost+LOW_BOUND(U)$\geq$    // Performs traditional branch and bound:
       $BestCompleteSolutionSoFar$)           // if current path is known to take longer
         return TRUE                          // than a previous solution

// NOT_FILLED: looks for an unscheduled-but-ready instruction that can schedule with IG
   if ($\exists$ an instruction, $i \in L$, such SCHEDULABLE($IG + i$))
         return TRUE

// NOT_NEW: looks for cases where the last cycle was empty, and none of the scheduled
                                              // instructions are new
   if (($PreviousIG = \emptyset$) and ($IG \neq \emptyset$) and ($\nexists$ an instruction, $i \in IG$ such that $i \in NewR$))
         return TRUE

// SAME_ECLASS:checks whether the e-classes of all of the instructions of IG match to the
                                              // e-classes of a previously tried case
   if ($\exists$ previous schedule, PS, such that, for $\forall$ I $\in$ IG, $\exists$ a corresponding I' $\in$ PS, where
       ECLASS(I)=ECLASS(I') and where I' is only used by a single I)
         return TRUE

// STRICTER_ECLASS_WORKED:checks whether the e-classes of all the instructions
// of IG can be matched to correspondingly stricter e-classes in a previously successful schedule
   if ($\exists$ previously successful schedule, PS, such that, for $\forall$ I $\in$ IG, $\exists$ a corresponding I' $\in$ PS,
       where ECLASS(I) $\geq$ ECLASS(I') and where I' is only used by a single I)
         return TRUE
   return FALSE
end
```

Figure 5.2: Pruning techniques that run in linear time

as well as from their resource requirements. The lower bound on the code-size cost of the remaining instructions is the number of long-words needed to schedule the remainder of the trace, and is defined as the largest of *three separate lower bounds.* The first is based on resource requirements. For example if a particular machine only allows two floating point instructions per cycle, and if there are 7 floating point instructions remaining to be scheduled, then at least 4 long words will be needed. The second is based on the overall issue width. In a 6-wide VLIW, for example, scheduling 13 instructions requires at least 3 long-words. The third lower bound is the number of cycles on which one or more instructions from any critical path in the remaining DAG must be scheduled. A critical path in a DAG is any path of maximum latency. For a schedule of minimum run-time, a critical path instruction has no choice of when to be scheduled: the cycle when it becomes ready is also its deadline. Such instructions with no scheduling flexibility are called FIXED_CYCLE instructions. Assigning the maximum of the above three lower bounds to *#Long_Words_LB*, we find a lower bound on the NOP cost by: *NOP_LB = (VLIW_width * #Long_Words_LB) - #unscheduled_instructions.*

The second method of Figure 5.2 is NOT_FILLED pruning. To see its motivation consider a yet-to-be-scheduled instruction, A, that is not in the currently-chosen-to-be-scheduled set of instructions, IG (*i.e.,* $A \in R, \quad A \notin IG$). If after scheduling IG there are unfilled slots (NOPs) in its long-word, then it never sacrifices optimality to schedule A in one of those unfilled slots rather than leaving it empty. Thus schedules in which A is not scheduled can be pruned. A similar argument can be constructed for multi-NOP or EPIC architectures.

The third pruning method is NOT_NEW. Its intuition is that the SCHEDULE_RECURSIVE procedure of Figure 4.1 is called per cycle, and different cycle-by-cycle search sequences in the algorithm can correspond to the same code. Figure 5.3 shows an example of how this can happen in a 2-wide VLIW. For the DAG shown in Figure 5.3(a), Instruction #2 can be scheduled on any cycle between 2 and 10. The resulting code is the same, however, and is shown in Figure 5.3(b). In such a case, we prune search paths where no new instructions are chosen on the current cycle and where the previous cycle had no instructions scheduled. This is because

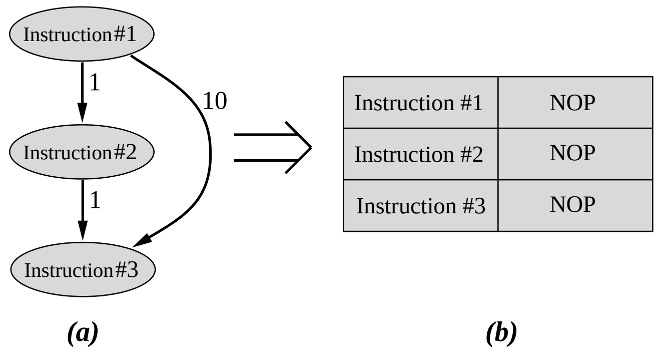


Figure 5.3: An example of where scheduling an instruction on different cycles results in the same final code.

the search path where the same available instructions were chosen in the previous cycle to schedule would have yielded the same code possibilities as the current path. (For understanding Figure 5.3, we remind the reader that an instruction may have two different outgoing latencies due to anti-dependence.)

The fourth pruning method is SAME_ECLASS. It stems from the observation that sometimes instructions are equivalent from the viewpoint of scheduling. If two instructions use exactly the same resources and contain exactly the same outgoing DAG edges (with the same weights on the edges), then they belong to the same equivalence class (or e-class). Thus, if one of these instructions was already tried as a member of IG, then there is no need to try the other one separately. While the code shown in Figure 5.2 does achieve this (by only allowing the selection of the first one), it is only included for ease of presentation. In practice, this type of pruning is more easily achieved by modifying the for loop of Figure 4.1 so that it loops on equivalence classes, e-classes, rather than on instructions. One nice feature of e-classes is that the computation of which classes exist in a trace can be performed just one time, prior to the recursive search in Figure 4.1. For our test machine, we measured an average of 1.3 instructions per e-class. While this number is not large, the exponential nature of the search space makes this optimization worthwhile.

Finding a low-overhead implementation of SAME_ECLASS pruning requires some thought. First, we define NC_{left} to be the set $READY$ instructions that are not chosen for the current instruction group, and we also recall the previous definition of NC_{subset} to refer to the set of instructions that belong to the current instruction group, IG, and that are not critical, as indicated in Figure 5.1. We also define the notion of an *earlier* instruction among the set of ready-to-schedule instructions, R. One instruction is considered to be "earlier" than another if if it appears first in the original program order. Although such an ordering is arbitrary – the elements of R being mostly independent – it allows us to identify when a previous schedule has been tried.

The most obvious (but overly-expensive) method for accomplishing SAME_-ECLASSES pruning is to, for each member, I_{NC}, of NC_{subset}, search for an earlier instruction of the same type within NC_{left}. If such an instruction is found within NC_{left}, this instruction must be further checked to verify that it does not have a zero-cycle dependency from any other element of NC_{left}. When these conditions are met, we may safely prune the current instruction group, IG. We do not need to verify that the new instruction is schedulable (that is, we do not need to call the FITS_IN_ONE_LONG_WORD function), because it is of the same type as I_{NC}.

The more-efficient method employs bit vectors. We define four new bit vectors: 1) the instructions that are elements of the Ready set, BV_R, 2) the instructions that are elements of the Chosen set, BV_{IG} 3) the instructions that belong to the Not-

Critical set and also have no zero-cycle dependencies from any other element of R, BV_{no_dep}, and 4) an array vectors, one for each e-class, that identifies its members, $BV[e-class]$. The first three of these bit vectors already exist in our implementation for other purposes, so that the there is no computation cost for using them. The fourth of these, the array of bit-vectors for each e-class, is not dependent on the current scheduling choice, so that it may be pre-computed prior to the exhaustive search.

Figure 5.4 presents the full algorithm for SAME_ECLASS pruning. In this algorithm, each of the instructions that might potentially be replaced, I_{NC} are each examined in turn. Those instructions that have zero-cycle edges to other chosen instructions are not considered further, however, because the SAME_ECLASS pruning method only considers replacing individual instructions, and the removal of this instruction would mean that another of the chosen instructions will also have to be removed. The main difference between Figure 5.4 and the less-efficient method described above is that the elements of the not-chosen set are not individually examined. Instead, two bit-vector subtractions are used to reduce the not-chosen set into the set of true replacement-candidates. One of these subtractions removes all of the instructions that have zero cycle dependencies from other members of the Ready set. There are two reasons why these dependencies might cause problems: 1) if the instruction that it depends on is not scheduled, and 2) if the it depends on is scheduled but the ordering restriction prevents this instruction from occupying I_{NC}'s slot. While this approximation is conservative we note two facts: 1) that it is rare, and 2) that it does not harm our optimality guarantees, because it only

```
SAME_ECLASS ($NC_{subset}, BV_R, BV_{IG}, BV_{no\_dep}, BV[e-class]$)   // Inputs: the set of
// not-critical-but-chosen instructions, a bit-vector of ready instructions, a bit-vector of
// chosen instructions, a bit-vector of ready instructions that are not dependent on
// others, and an array of bit-vectors for the elements of each e-class

    define $BV_{left} = BV_R - BV_{IG}$          // A vector of ready-but-not-chosen instructions
    for each $I_{NC} \in NC_{subset}$
        if (HAS_DEPENDENCIES_TO_OTHERS($I_{NC}$,$IG$))        // Cannot remove these
            break
        end if

        define $BV_{cand} = BV_{left} - BV_{no\_dep}$   // Slightly conservative, prevents considering
                                                     // instructions that depend on other members

        if ($BV_{cand} \geq 2^{POSITION(I_{NC})+1}$       // Identifies whether $I_{NC}$ occurs earlier than
            return TRUE                              // at least one of the replacement candidates
        end if
    end for
    return FALSE
end
```

Figure 5.4: An efficient implementation of the SAME_ECLASS pruning method

reduces the amount of pruning. The final insight of the algorithm in Figure 5.4 is that *a $\geq$ comparison is used to identify whether any of the replacement candidates occurs later than I_{NC}*. Finding a later candidate is equally as effective as finding an earlier one, because all members of the same e-class have identical output dependencies. In addition, the $\geq$ comparator allow us identify whether any of the higher bit positions is set – *without the need to search them*; there is no analogous operation for finding an earlier candidate – the $<$ comparator fails because candidates may simultaneously exist in both higher and lower positions.

The fifth pruning method is STRICTER_ECLASS_WORKED. For an e-class of instructions, E1, to be strictly harder to schedule than another e-class, E2, the following two conditions must hold. First the instructions of E2 must be schedulable with the resources used by E1. This can happen, for instance, when E1 is a subtype[1] of E2, which by definition means that an E2 instruction can execute on the same functional unit as an E1, but not necessarily visa-versa. Second, the instructions of E2 must contain no out-going DAG edges that the E1 instructions do not also contain, and none of the outgoing edge latencies of E2 can be larger than the corresponding latencies of E1. If both of these conditions are met, we can say that a schedule using an E1 is always preferable since E1 is harder to schedule, and the schedule using E2 can be pruned. For our test programs, each e-class tends to have one stricter e-class.

Efficient identification of valid schedules containing stricter e-classes of an instruction, I_{IG} is more difficult than the identification of schedules with instructions of the same e-class as I_{IG}, because *there is no guarantee that that the stricter e-class is schedulable.* Rather what is guaranteed is that, *if* the stricter e-class is schedulable, then it is at least as good as the current choice that uses I_{IG} instead. As a result, the stricter instructions will each have to be tested for schedulability. Figure (A bit vector approach is also possible, based on the various *types* of the weaker instructions, but it was found to be less efficient.)

Figure 5.5 presents the STRICTER_ECLASS_WORKED algorithm. This algorithm makes extensive use of the FSA methods of Chapter 4, Section 4.2 to allow for efficient computation. The algorithm begins by considering each of the non-critical-but chosen instructions, I_{NC}, in turn. If any of these instructions contains

[1]Some VLIWs define subtypes, which are instructions with additional restrictions to those of their parent type. For instance a subtype of an integer operation might be only schedulable to the first IALU

```
**STRICTER_ECLASS_WORKED** ($NC_{subset}, FSA\_State_{critical}, BV_{left}, BV_S[e-class]$)
// Inputs: the set of not-critical-but-chosen instructions, the FSA state corresponding to just
// scheduling the critical instructions, a bit-vector of ready-but-not-chosen instructions, and an
// array of bit-vectors for each e-class, identifying all instructions belonging to a stricter e-class

   for each $I_{NC} \in NC|subset$
      define ($BV_{cand} = BV\_left$ & $BV_S[ECLASS(I_{NC})]$)  // bit-wise AND finds all stricter
                                                   // instructions within the ready-but-not-scheduled set
      if ($BV_{cand} == 0$)                        // No stricter instructions to test
         break
      end if

      SET_FSA_STATE($FSA\_State_{critical}$)       // Critical instructions are always scheduled
      for each $I_{other} \in NC_{subset}$
         if ($I_{other} \neq I_{NC}$)              // Include all but the current instruction into the FSA
            ADVANCE_FSA_STATE($I\_other$)          // Update the FSA
         end if
      end for
      for each $I_{stricter} \in BV\_cand$         // Consider all stricter instructions
         if (FITS_IN_FSA($I\_stricter$))           // This essentially calls FITS_IN_1_LONG_WORD()
            return FALSE                           // Stricter instruction fits
         end if
      end for
   end for
   return FALSE
end
```

Figure 5.5: An efficient implementation of the SAME_ECLASS pruning method

stricter-eclass-candidate instructions within the ready-but-not scheduled set, BV_{left}, then these elements must be tested in a two-step process. First, the *FSA_State* without instruction I_{NC} is constructed by adding the other non-critical-but-chosen instructions into the *FSA_State* that corresponds to only scheduling the critical instructions. Second, each candidate instruction, $I_{stricter}$, is then tested for schedulability using this FSA. (Intra-cycle dependency hazards all also checked, as described in the supplemental Section 4.3 of Chapter 4.)

5.2 Identifying optimal schedules quickly

Where as the five methods in Figure 5.2 identify *bad* choices to skip, it is also possible to identify *good* solutions and quit early (without considering the remaining combinations of C). If a given solution is provably optimal (*i.e.,* its cost equals the lower bound) then there is no need to search for a better solution. This optimization is indicated in Figure 4.1 by means of the early return from inside the loop.

Two types of situations allow for the detection of optimal schedules; they correspond to identifying the globally-optimal schedule and locally optimal schedules. A complete schedule is globally optimal if its code size matches to the lower bound on the total code size of the entire trace. (This lower bound is found using the methods described in Chapter 4.) Similarly, a schedule is locally optimal code size *of the remaining instructions* is equal to the lower bound for these instructions. In the case of finding a globally-optimal schedule, the trace is finished scheduling. In the case of locally optimal schedules, the *current* level of recursion may finish.

One additional optimization is used to increase the effect of optimal-detection pruning of leaf nodes. We observe that, if all of the remaining instructions are not schedulable within one long word, then at least two long-words will be needed. When only a few instructions remain to be scheduled, this observation may lead to a more accurate lower bound, if the existing bound-techniques only predict a single cycle. For instance, if only 5 instructions are left, but we choose not to schedule all of them, it could only mean that they are not all schedulable (otherwise NOT_FILLED pruning would have prevented the current choice). Therefore, we can predict a minimum of 7 NOPS (6-wide VLIW $\times$ 2 cycles - 5 instructions). Although this optimization only works leaf-nodes, we note that much of an exhaustive search's execution time is spent in such leaf nodes.

Search order Beyond all of these optimizations, the efficiency of branch and bound algorithms is known to depend upon the search order – if the optimal solution can be found early in the search, then the rest of the choices do not need to be evaluated. Search order is determined in Figure 5.1, by the line reading: "**for each** NC_{subset} combination of elements of *NC*." This line indicates that all combinations are tried, but it does not specify the ordering algorithm. Figure 5.6, reproduces the algorithm of Figure 5.1, but with the aforementioned line changed. Comparing the two figures, the Figure 5.6 contains four new lines, which specify: 1) the maximum number of non-critical instructions; 2) the assigning of an initial value to a bit-vector, BV, where each member of NC is associated with one bit from BV; 3) a call to an as-yet-unspecified function, NEXT_CHOICE(), that determines the search order; and 4) a line to create the NC_{subset} from the current bit-vector choice, BV.

Among these modifications, the crucial question is the nature of the algorithm used in the NEXT_CHOICE() function. One simple algorithm to search all possible orderings is to create a loop such as: "**for** $(BV = 0; BV < 2^{SIZE(NC)}; BV{+}{+})$." The problem with this solution is that there will be little variation from one combination to the next, where as we find that the search space can be checked more efficiently when the combinations that are tried have a high degree of variation.

Therefore, a better search method is to use a nearly-random search order. We

```
SCHEDULE_RECURSIVE(U, PrevIG, PrevR)            // U:the set of not-yet-scheduled
                // insts, PrevIG: insts scheduled on last cycle, Prev R: insts ready on last cycle
  ADVANCE_CLOCK(Clk)                            // Advance the clock
  define Best = MAXINT                          // Initially, no best solution
  if ($U = \emptyset$)                          // See if finished
    return 0
  define $R$ = READY($U$)                       // The set of ready-to-schedule insts
  define $NC$ = NOT_YET_CRITICAL($R$,Clk)       // All R that could be delayed
  define $C = R - NC$                           // All R that must schedule now
  define $Max = VLIW\_WIDTH - SIZE(C)$          // Max # of non-critical instructions
  define $BV$ = INITIAL_CHOICE()                // Choose the first combination
  while NEXT_CHOICE($BV$, $NC$, $Max$)          // Employ the search-order function
    define $NC_{subset}$ = MEMBERS($BV$)        // convert bit-vector
    $IG = C + NC_{subset}$                // this inst group: all critical + some non-critical
    (Ccost) = FITS_IN_1_LONG_WORD(IG)           // Finds cost of current selection
    if ( not PRUNE(Ccost,U,R-IG,IG,PrevIG,PrevR)) // Skip unreasonable ones
      Rcost= SCHEDULE_RECURSIVE(U-IG,IG,R)      // Cost of rest (U-IG)
      cost = Ccost + Rcost
      if (Best > cost)                          // Is this solution the best so far?
        Best = cost
      if (Best = Ucost)                         // Is this a minimum-cost solution?
        return Best
    end if
  end for
  return Best                                   // All possibilities have been explored
end
```

Figure 5.6: Optimal algorithm with scheduling order specified as a function call.

cannot use a fully-random algorithm because not only are calls to the random number generator expensive (and these calls would be frequently executed), but also because we wish to avoid repeating combinations. Fortunately, the *pseudo-random shift register* exactly solves both of these problems, providing a way to *quickly* generate nearly-random numbers than *never repeat* choices until all combinations have been tried. The workings of the pseudo-random shift register can be found in [62], among other places; here we will only describe its more general behavior. Given a pseudo-random number, BV_i, the next pseudo-random number, BV_{i+1}, is found by shifting BV_i "up" by one bit – the high-order bit being clipped to the length of the bit vector and the low-order bit position being filled by the exclusive-OR of a small

```
bit_vector_type INITIAL_CHOICE()          // The initial choice is just the empty set
  return 0
end

boolean NEXT_CHOICE($BV$, $NC$, $Max$)    // Find the next choice based on the current
  if ($BV$==0)                            // The initial value is special
    if (First_Time)                       // On the first iteration, keep 0
      return TRUE                         // Returning TRUE means its a new combination
    $BV = 2^{Max+1} - 1$                  // After 0, we fill up the maximum # of bits
    return TRUE                           // This is a new combination
  end if

  while(1)                                // Doesn't run forever but goes back to 1st choice
    $BV$ = Pseudo_Random_Shift($BV$)      // Shifts, XORs top bits, see [62]
    if ($BV == 2^{Max+1} - 1$)            // Has it come back to the first value?
      return FALSE                        // All combinations have been searched
    if (SIZE(BV)$\leq Max$)               // This is low cost, because we just have to keep
      return TRUE                         // track of if a bit went into bottom or out-of top
  end while

end
```

Figure 5.7: The INITIALIZE_CHOICE() and NEXT_CHOICE() functions that are used by Figure 5.6.

set of upper bits. If this set of upper bits is properly chosen, then the value, BV_i will not repeat until all other numbers (except zero) have been visited. This property implies that $BV_i = BV_{i+2^{SIZE(NC)}-1}$.

In using the pseudo-random shift register, we perform two modifications. First, we consider the "zero" choice before using the shift register. Second, we speed up the search by skipping choices that contain more instructions than can fit within the VLIW width.

Figure 5.7 presents the final algorithm. This algorithm treats zero as a special case. Once the zero choice has been tried, it next chooses $BV = 2^{Max+1} - 1$; the idea is that the shift register has to be started with some value, and so we start with an aggressive choice (*i.e.,* a choice that tries to fill as much as possible). On remaining iterations, the pseudo-random-shift-methods of [62] are used. This part of the algorithm is contained within a while loop, which terminates by returning either when a possible combination is found, or all choices have been considered. The purpose of the while loop is to skip choices that contain more bits than can be scheduled. The insight is that we do not need to count up all bits of BV to know if it exceeds the maximum allowed size, Max. Instead, since we already use the top bit in computing the bottom bit, we can easily know size of BV, by simply adding this upper-and-lower bit information to a running total; decrementing if the old top-bit was "1" and incrementing if the new bottom-bit is "1".

Chapter 6

Non-optimal heuristics to reduce compile-time

Although branch-and-bound greatly reduces the search space without sacrificing optimality, the compile time for some large or complex traces is still unmanageable. In such cases, non-optimal heuristics must be used. Such approaches reduce the compile time by only searching a small portion of the solution space, without guaranteeing optimality. We use such heuristics only for larger traces that have already taken a long time to compile. A desirable consequence is that smaller traces, and easy-to-schedule larger ones, are scheduled optimally. We investigated many heuristics, and identified five that were effective.

Splitting Large Traces Since the search space grows exponentially with the trace length, the longest traces must be split into *trace-chunks*, each being of a fixed length. (The final chunk is an exception to this rule, since it contains the leftover instructions, which will usually not extend to the full allowed length.) We empirically determined 46 instructions to be a reasonable chunk size. Since most traces are shorter than 46 instructions, they will use a single trace-chunk, Therefore, when using the term *trace-chunk*, we do not necessarily imply that the original trace has been split into separate pieces.

To effectively split traces into chunks, one must compensate for two effects that are seen at the boundaries between chunks; the first of these boundary effects is a reduced parallelism near the edges of trace-chunks. While splitting has the general disadvantage of reducing the available pool of instructions within which to identify parallelism, this disadvantage is most severe for those instructions near the boundary between trace-chunks. This occurs because parallelism is often found among instructions that lie close to each other in the original program order, so that instructions near a boundary will tend to have a greater degree of parallelism with instructions from the neighboring trace-chunk. The solution is to re-schedule the *tail instructions* of the previous trace-chunk as a part of the next trace chunk. The tail portion is defined as those instructions that our methods have assigned to the bottom N long-words of the chunk's schedule, where N is chosen to be as large as possible such that the total number of instructions present in these bottom long words is less than an *Overlapping Fraction* of the trace-chunk size. We use $\frac{1}{3}$, implying that at least 31 instructions are committed by the previous trace-chunk (since two-thirds of 46 is 31). We also note that these tail instructions need not have been near the bottom of the trace-chunk in the original program order. We also note that, by rescheduling the tail instructions,in the next chunk, these instructions have the opportunity to either keep their original assignment, or to improve upon it by using instructions from the new trace-chunk.

The second boundary effect is cross-boundary data dependencies. An instruction, I_1 that is committed within one trace-chunk could have a multiple-cycle dependency to an instruction, I_2, in the next chunk. If each trace-chunk is simply scheduled independently, such a dependency might cause unnecessary stall cycles by scheduling I_2 too close to I_1. To avoid these stalls, we compute

```
SCHEDULE_CHUNKS(Trace, SplitSize, OverlapFraction)  // Inputs: the trace, the length
                                  // of each chunk, and the amount of overlap between chunks

  do
    define Chunk = ∅
    for each Instruction ∈ Trace        // Visit each instruction in original program order
      if ((SIZE(Chunk) < SplitSize) AND (NOT_ALREADY_COMMITTED(Instruction)))
        Chunk += Instruction
      end if
    end for

    COMPUTE_DFG(Chunk)
    if (NOT_THE_FIRST_CHUNK(Chunk)//First chunk has no cross-boundary dependencies
      INSERT_DFG_EDGES_FOR_DEPS_FROM_COMMITTED_INSTRUCTIONS(Chunk)
    end if

    SCHEDULE_RECURSIVE(Chunk)     // Schedule Chunk using our methods

    for each Instruction ∈ Chunk          // Visit each instruction in new schedule order
      if ((POSITION(Instruction) < OverlapFraction × SplitSize) OR      // Upper portion
          (HAS_ALL_REMAINING_INSTRUCTIONS(Chunk)))//Final chunk has no overlap
        COMMIT(Instruction)
      else if (BELONGS_TO_SAME_LONG_WORD_AS_LAST_COMMITTED(Instruction)
        COMMIT(Instruction) // All instructions within one long-word commit together
      end if
    end for
  while SOME_ARE_STILL_NOT_SCHEDULED(Trace)
end
```

Figure 6.1: Algorithm for selecting and scheduling trace-chunks, including our optimizations for boundary effects.

the earliest possible cycle for I_2 to be scheduled as $Earliest(I_2) = latency(I_1 \rightarrow I2) - (Last\ Cycle\ of\ Previous\ Chunk - Scheduled\ Cycle\ of\ I_1$, and then we insert a new DFG edge to I_2 with a latency $Earliest(I_2)$ from the top of its trace-chunk.

Figure 6.1 summarizes our algorithm for splitting traces into smaller chunks. In this figure, three details a worth noting. First, trace-chunks are built based on the instructions' original program order, but are committed based on the instructions' new ordering. Therefore, an instruction from the first trace-chunk could theoretically be repeatedly passed down and ultimately be scheduled in the final chunk. Second, the BELONGS_TO_SAME_LONG_WORD_AS_LAST_COMMITTED() function prevents the splitting algorithm from committing only part of a long-word. Third, the HAS_ALL_REMAINING_INSTRUCTIONS() function enable the final chunk to commit all of its instructions. As a result, *any trace with 46 or fewer instructions will not be split,* leading to a more-optimal result.

Resource Contention on FIXED_CYCLE Instructions Our second non-optimal heuristic is based on the properties of FIXED_CYCLE instructions. Recall that a FIXED_CYCLE instruction can only become ready on the same cycle as its deadline; so, the exact clock when it will schedule is known in advance. As a result, it is possible to predict resource contention among these fixed instructions. For example, if there are three FIXED_CYCLE floating point instructions with the same deadline, they cannot be scheduled together on a machine with two floating point units. Since a schedule will not be found that respects all instruction deadlines, the default mechanism is to simply let the scheduler try to find a solution, eventually fail, and then, using the approach of Chapter 4, automatically increasing the instruction deadlines by 1, and re-running the scheduler. A better, and still optimal, pruning strategy would be to increment all of the instruction deadlines prior to ever scheduling, since such a scheduling attempt is guaranteed to fail. In practice, however, this was found to greatly increased the search space by considering many unwise choices. Instead, we place an edge of latency 1 between conflicting FIXED_CYCLE, and then repeat the scheduling. This approach is effective since the scheduling bottleneck is being directly tackled, while it is still often able to find the optimal solution.

Quick Cycle-Count Estimates Our algorithm attempts to find the optimal-for-code-size schedule, from among those that are optimal in run-time. As a consequence, it is first necessary to identify what the optimal run-time of a trace-chunk is, which is exactly the task for which traditional schedulers have been designed; it is, by itself, an NP-complete problem. Without knowing the optimal run-time, therefore, our default method instead employs a lower bound estimate, as described in Chapter 4. If no schedule can be found, then the estimate is incremented, and the search is repeated.

Since searching for schedules where none can exist will obviously increase the compile time, we introduce a cycle estimate heuristic. To begin with, even though instruction scheduling for run-time is NP, it is still a much simpler problem than instruction scheduling for run-time *and* code-size. In fact, many compilers have taken a greedy approach to this problem, finding that the results are nearly optimal. Therefore, it is reasonable for our method to employ a similar approach to the same problem.

Our approach to deriving a quick cycle-count estimate for a trace-chunk has two parts. To begin, we employ a basic greedy scheduler, like many current methods. Next, we also implement a random-schedule optimization to refine this cycle-count estimate. A random schedule does not employ compile-time-expensive exhaustive-search techniques; instead, it simply follows one randomly-chosen path in this search space (as also described in Chapter 5 – the difference being that Chapter 5 uses random schedules for *code-size* without loss of optimality, where as here they are being used for a non-optimal, *run-time* estimate). Since the potential search space increases with trace-chunk size, we choose to compute as many random schedules as there are instructions in the trace-chunk. Comparing the run-times of these random schedules, with that of the greedy schedule allows for a somewhat-more-accurate initial estimate.

This initial run-time estimate is then used by our run-time-and-code-size instruction scheduler. In this way, we avoid the compile-time cost of looking for a solution where none exists – since this run-time estimate is based on the observed run-time of one of the above schedules, then we know that there exists at least one schedule with this cycle count. Avoiding this compile-time cost comes at the expense of lost optimality, however, because the final solution is no longer guaranteed to be optimal in run-time. Yet the effect is slight not only because the run-time estimate is usually correct, but also because our scheduler *may* still find a solution smaller run-time if one exists and happens to also have a smaller code size.

Timing Out Through our pruning methods, most traces quickly arrive at good schedules; yet there remain a few difficult traces that may compile for hours without ever finishing. For these traces, it is necessary to implement a time-out heuristic. Whenever the scheduling of a trace-chunk takes longer than a preset amount of time, it will time out, taking the best solution found so far. We have found that, even though these trace-chunks would require hours to compile without a time-out, yet the time-out value may be much smaller without causing a significant performance penalty. In fact, as detailed in the results, when the time-out value is increased from 1 second to 100 seconds per trace-chunk, the code size only improves by 0.09% on average.

The smallness of this improvement is due to two factors. First, even though a trace-chunk requires a long time to compile, it in many cases actually finds the best solution much earlier. This situation arises from the branch-and-bound nature of the algorithm; during the search process, one cannot know whether the best solution found so far is the overall best solution until all remaining choices have been compared against it. Since there are often many solutions of equivalent cost, it follows that the optimal solution is likely to be found early in the process. Second, the best solution may require longer than 100 seconds to be reached. This case is both rare and not-easily avoided, however.

The presence of our hard time-out function allows the user to bound the compile-time. In the worst case, if there are N_{TC} trace-chunks, and if every one if them times out, then our method would suffer a compile-time overhead of N_{TC} seconds (assuming a 1 second time-out). Since N_{TC} is fixed and known in advance, the user may adjust the time-out value to meet his compilation needs.

In reality, very few trace-chunks require a full second to compile. In fact, simple traces having only a few instructions will consume almost no time, because the search space is so small. Therefore, we have extended our time-out method to provide a variable compilation time-limit for less complex traces, while still maintaining an upper bound of 1 second. The new time-out value uses an empirically-determined function that considers the trace-chunk's length, and the amount of flexibility within the DFG, as described in the results. Using this approach, we find that, not only will all trace-chunks complete within 1 second, but that the simplest traces are *guaran-*

teed to finish within 5 milliseconds. The time-out function for simple trace-chunks is chosen so that it has almost no effect on the code-size or run-time results. Interestingly, it also has little effect on the compile-time; its purpose is not to improve the compile-time but only the *worst-case compile-time guarantees.* Using this approach, we find that, on average, our benchmarks can be guaranteed to finish within a factor of 6 of the original compile time.

Eager-Lazy We have found that, rather than abruptly halting the search when the time runs out, it is better to provide for a more gradual approach. We note that, since our branch-and-bound algorithm employs a depth-first search, the leaf nodes are often visited, but the root node will only change infrequently. As a result, when the time-out occurs, it is possible that only one schedule of the first long word may have been considered – meaning that other portions of the search space have never been explored.

To cover more of the search space, we therefore introduce an eager-lazy optimization similar to the one that we first proposed in [14]. In that previous study, we considered instruction scheduling for IA-64 machines. Unlike the current work, [14] targets EPICs rather than a fixed width VLIW. And [14] ignores resource constraints; instead it provides only a lower bound on the code size, based on the templates provided by the IA-64 ISA. In particular, long words may be of an *unbounded* length.

Despite these differences, [14] presents an eager-lazy approach that applies, in modified form, to fixed-width VLIWs; we now describe this modified method. Using the syntax of Figure 4.1 each scheduling cycle contains a set of *ready* instructions, R, a subset of which are *critical*, C. Instead of attempting all possible combinations of the set R eager-lazy only considers two combinations: R (eager) or C (lazy). Such an all-or-nothing approach works for the general IA-64 ISA, but on a fixed width VLIW, neither may be a good choice. For one thing, R may contain more than 6 instructions; to address this issue, our modified algorithm replaces the eager selection of R with the eager selection of *as much of* R *as fits into one long word.* For another thing, scheduling only C may leave empty NOP slots, exactly what our NOT_FILLED pruning method of Chapter 5 aims to avoid. Therefore, when $C \neq$, we instead choose a second, *different* eager selection.

Since we only search two combinations per level, the compile-time will be faster, but there will be no optimality guarantees, and the E-class pruning techniques must be disabled, because they assume that all schedules from the set R will be considered. Since optimality is not preserved, we only wish to employ this eager-lazy method after first giving the scheduler a chance to find an optimal solution. *Therefore, we divide the compilation time into two parts.* Complex traces are given $\frac{1}{2}$ seconds for exhaustive search, and then the remaining $\frac{1}{2}$ second uses the eager-lazy approach. This achieves some of the advantages of each method: maintaining optimality for most trace chunks, while more exploring a more diverse portion of the search space.

Chapter 7

Instruction scheduling across basic blocks for code size

Up to this point, we have considered the problem of instruction scheduling within a single basic block, however instructions may be scheduled across basic blocks. Historically, such scheduling techniques have been used to increase performance at the expense of code size. In contrast, our approach reduces both the run-time and the code size. This is the first time that such across block instruction motion has been used to reduce code size.

The outline of this chapter is as follows. Section 7.1 describes the trace identification algorithm of [2]. Section 7.2, then discusses how code-size considerations may affect the trace identification algorithm. Next, Section 7.3 considers the issues of compensation code in traditional trace scheduling. Finally, Section 7.4, introduces our use of traces in a code-size-reducing instruction scheduler.

7.1 The algorithm for trace identification

Trace identification is an involved, three-phase process in [2]. First, profiling is used to identify the execution frequencies and branch probabilities of all basic blocks in each function. For this purpose, we have implemented a profiling mechanism into our compiler infrastructure. Second, among the basic blocks which have not been assigned to a trace, the one with the highest frequency is chosen to starting a new trace. This new trace begins with the chosen block and grows by iteratively adding the most likely predecessor *or* successor until the next addition would cause the *trace probability* to fall below a user-defined value, $threshold_{trace}$ (75% in our experiments). Cyclic edges are also not permitted within a trace. Third, the second step is repeated until all blocks are part of some trace.

The preceding discussion introduces the concept of *trace probability.* We now formally define the computation of this probability. First, a trace consists of an ordered set of basic blocks, $b_1, b_2, ..., b_n$. We may define $P(b_{i+1}|b_i)$ to be the probability that b_{i+1} will execute, given that b_i executes. In a complementary manner, let us define $P(b_i|b_{i+1})$ to be the probability that b_i would have been the previous block to execute, given that b_{i+1} is being executed. These definitions can be extended to describe a chain of blocks, $b_1, b_2, ...b_n$: $P(b_1|b_n) = P(b_1|b_2) \times P(b_2|b_3) \times ... \times P(b_{n-1}|b_n)$ and $P(b_n|b_1) = P(b_n|b_{n-1}) \times P(b_{n-1}|b_{n-2}) \times ... \times P(b_2|b_1)$. Finally, we may define $trace\ probability(b_1, b_2, ..., b_n) = Min(P(b_1|b_n), P(b_n|b_1))$.

Figure 7.1 is a helpful example in understanding the trace selection algorithm. This figure illustrates the control flow graph of a function with seven basic blocks. In Figure 7.1(a), the execution frequencies of all paths are indicated by the edge weights, as found through profiling. From this graph, the execution frequency of any basic block can be found either as the sum of all incoming edges, or as the sum of all outgoing edges.

To apply the algorithm for identifying traces, we first order the Basic Blocks according to their execution frequency: BB #1, BB #3, BB #6, BB #7, BB #5, BB #2, and BB #4. Therefore, the first trace will start with BB #1. Since BB

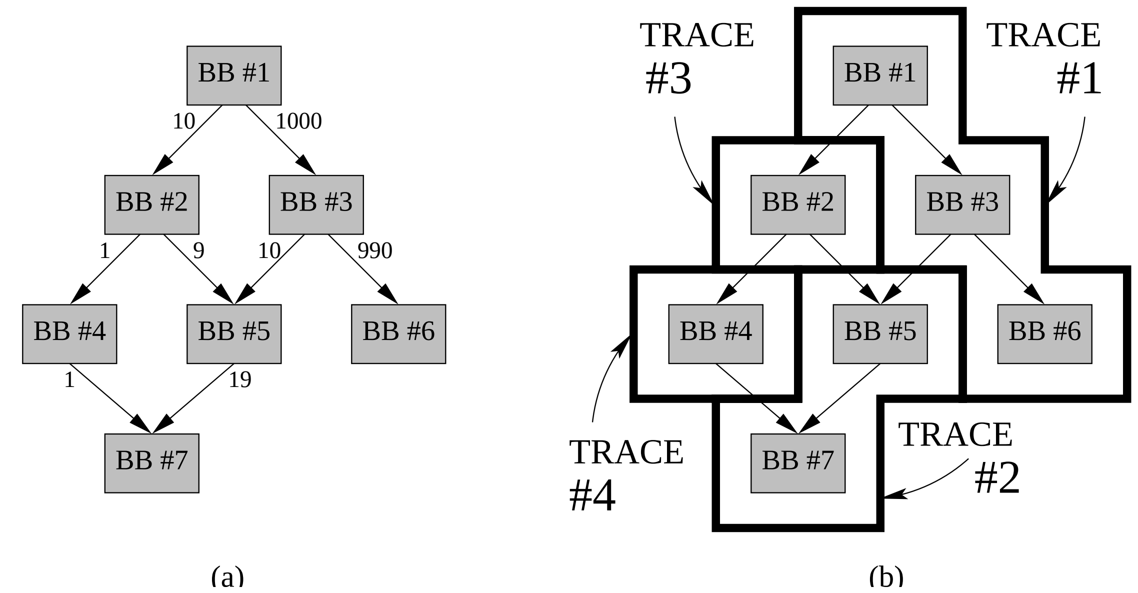


Figure 7.1: Trace identification. Through profiling, it is determined that a particular function is called 1010 times during the execution of the benchmark it belongs to. This function has seven basic blocks. In (a), the edges between the basic blocks are weighted by the number of times that the corresponding path is taken. Hence, BB#1 is executed 1010 times, BB#2 - 10 times, BB#3 - 1000 times, BB#4 - 1 time, BB#5 - 19 times, BB#6 - 990 times, and BB#7 - 20 times. In (b), the four traces of this function are indicated.

#1 has no predecessors, it can only grow down. The most like successor is BB #3. By examining the outgoing edges from BB #1, we see that $P(BB\#3|BB\#1) = 99.01\%(1000 \div 1010)$. Similarly the single incoming edge to BB #3 tells us that $P(BB\#1|BB\#3) = 100\%$. Since both of these probabilities are well above a $threshold_{trace}$ of 75%, BB #3 will be added to the trace. In the same way, BB #6 will be added next.

The second trace will begin with BB#7, since BB #3 and BB #6 were added to the first trace. The second trace will also include BB #5, because $P(BB\#7|BB\#5) = 100\%$, and $P(BB\#5|BB\#7) = 95\%$. The third trace will include only BB #2, and the fourth will contain BB #4. These four traces are indicated in Figure 7.1(b). From this example, we see that every block belongs to exactly one trace – although some traces are single blocks.

7.2 The impact of code size upon trace selection

When scheduling for code-size, the trace selection method of [2] may be extended in three ways. First, scheduling for code size allows for “call” instructions within traces. In traditional trace scheduling, there is little benefit from extending traces beyond basic blocks that end with “call” instructions. The reason is that, by the time the CPU returns from the called function, all values from the previous block are very likely to be available. In other words, there are no unresolved data-dependencies across calls. Of course, there could be data dependencies to the called function, but instruction schedulers do not consider these.

In trace scheduling for code size, however, there is a benefit of allowing “call” instructions within traces, because migration may reduce the code size. Yet the inclusion of call instructions complicates the scheduler. One problem is that many instructions cannot migrate across calls. For example memory instructions cannot migrate, because the called function might change the memory state. Similarly, instructions that use caller-saved registers cannot migrate. Another complication is that data dependencies which cross call sites become variable: if migration cause the two instructions that share a data dependency edge, I_1 and I_2 to be placed in the same block, then the data dependency is needed; if the instructions do not migrate, however, then the data dependency must be removed. Failing to remove this dependency edge can unnecessarily delay I_2 – its operands are ready immediately following return from the call instruction, but the data dependence might artificially prevent it from scheduling on the first cycles of its block. We solve this problem by first removing the dependencies in two known-to-be-safe cases: 1) both I_1 and I_2 have dependencies to a call instruction that lies between them, 2) other dependencies along a path between these instructions render exceeds (or equals) the latency on the direct edge between them. In all other cases, we impose an artificial edge to one of the call instructions that lies between 1_1 and I_2, so that it is then safe to remove the direct edge. In choosing which call instruction to place the edges to, we first look for one that already has an edge to either I_1 or I_2 (but obviously not to both, because then this would already be a known-to-be-safe case.)

A second change that occurs to the trace selection algorithm is that a lower $threshold_{trace}$ probability may be used. In traditional traditional trace scheduling, the choice of proper threshold is a trade-off between the run-time improvements of longer traces and the costs of increased compensation. At some point increased compensation code can degrade performance, because traces are constructed under the assumption that the side paths are infrequently taken. In our trace scheduling scheme, however, we avoid compensation code, and thereby avoid many of its costs. This can allow for a lower value of $threshold_{trace}$. Lowering this value has little impact on run time, since the probability of executing the whole trace reduces. Rather, the motivation for lowering the $threshold_{trace}$ value is to reduce code size, through scheduling flexibility. This benefit is possible because, unlike run time, the code size improvement is not dependent upon the likelihood of execution – after all, the size of the final code is simply the sum of the sizes of each trace, regardless of whether the traces are frequent or infrequent.

It is not reasonable to lower $threshold_{trace}$ by too much, however. A smaller threshold will produce longer traces. Since each block may only belong to one trace, choosing too many blocks for the current trace will reduce the options for forming subsequent traces. And fewer options might lead to a performance loss. In our code size methods, we use a $threshold_{trace}$ of 70%, which is a modest decrease from our default value of 75%.

A third modification to trace selection involves re-selecting traces. Based on the above observation that code-size improvements are not dependent on the likelihood of a path being taken, it is natural to consider traces that consist of infrequent paths. The problem of such traces is that they will not reduce the run-time, which is the original motivation of trace scheduling. The solution is to first select traces using frequency information, and then, after all traces have been scheduled, to perform a peep-hole optimization. This optimization considers all *pairs of basic blocks* that share a control flow graph edge, but that were *not* chosen to belong to the same trace.

Such basic-block pairs are then treated as new traces, and scheduled by our within-trace methods; yet certain restrictions are needed for good performance. For one thing, the instructions cannot be rescheduled in such a way as to increase the run-time along the more-frequent, original-trace path. For instance, suppose that BB#1 and BB#2 originally formed a trace, and that there is also a control-edge between BB#2 and another block, BB#3. If one of BB#2's instructions, I, depends on an instruction from BB#1, then when scheduling the BB#2-BB#3 basic-block pair, instruction I must not move up to a point where its dependency from BB#1 would result in a stall cycle along the frequent path from BB#1 to BB#2. This situations is avoided by inserting a dependency edge from the top to instruction I, when scheduling the basic block pair. An analogous situation also arises when instructions are moved down to such a point as to introduce stalls to a successor block.

Another restriction on rescheduling these basic-block pairs involves the compile time. Since there are likely to be more basic-block pairs than there are original traces, the scheduling of these pairs can drastically increase the compile time. For this reason, we view the scheduling of these block pairs more as a quick peephole optimization than as a complete rescheduling. Quick compilation is achieved by aggressive use of our non-optimal heuristics. Split sizes are reduced to 24 instructions, eager-lazy scheduling is activated from the start, and each pair is only give 20 ms to attempt a reschedule. In addition, schedules that do not migrate any instructions between the blocks can be pruned away.

7.3 Code size increase from compensation code in traditional trace scheduling

The performance gains of across-block code motion have traditionally come at the expense of code size. In this section we first illustrate the reasons for compensation code. Next, we consider situations that may prevent the creation of compensation code (thereby preventing the code motion). Finally, we explore how

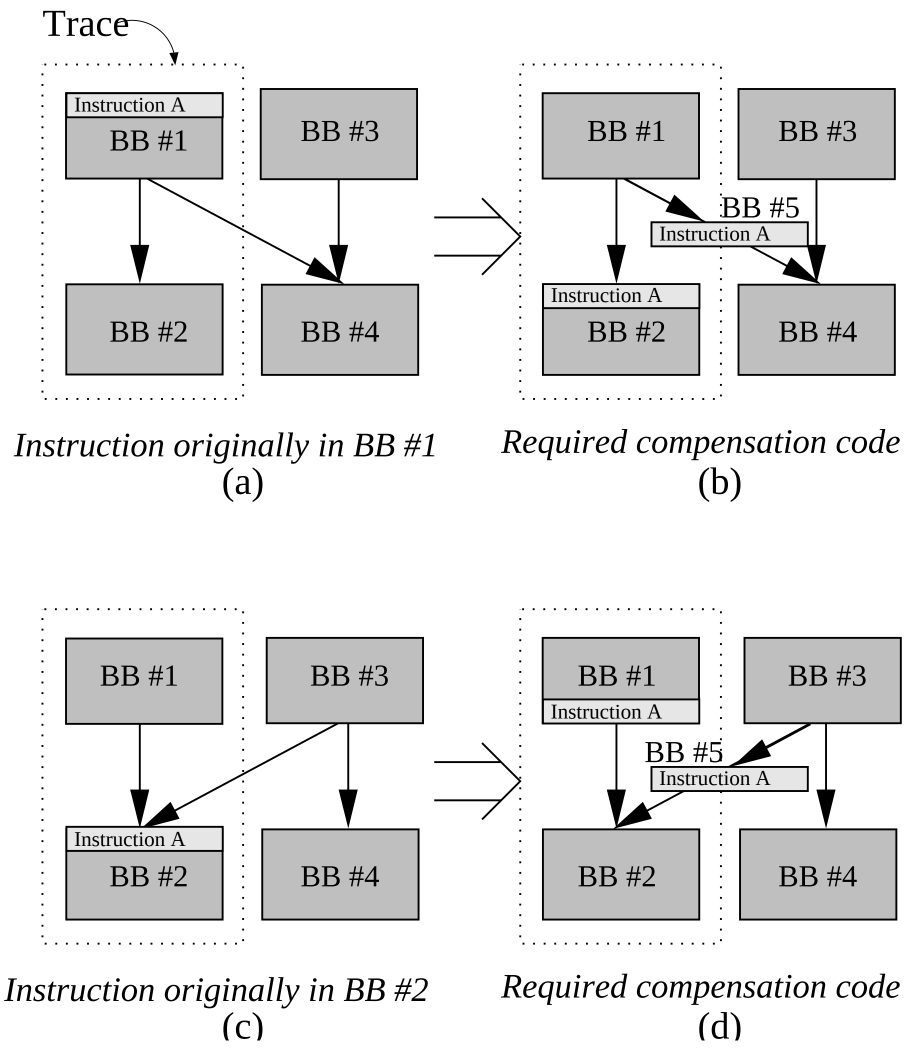

Trace
Instruction A
BB #1
BB #2
BB #3
BB #4
BB #5
Instruction originally in BB #1
(a)
Required compensation code
(b)
Instruction originally in BB #2
(c)
Required compensation code
(d)

Figure 7.2: Examples of the compensation code required to move an instruction below a side exit or above a side entry. The trace is composed of BB #1 and BB #2. BB #3 and BB #4 are not in the trace. In (a) *Instruction A* was originally in BB #1, above the side exit. In (b), the instruction is moved into BB #2, and compensation code has been inserted along the off-trace edge to BB #4. Similarly, (c) depicts an *Instruction A* that originally lies below a side entry, and (d) presents the compensation code that would be required to move it up.

to reduce the amount of compensation code.

Examples of compensation code Compensation code refers to the duplication of an instruction when it moves across a basic-block boundary. Figure 7.2(a) shows an example of why compensation is needed. In this figure, BB #1 and #2 are part of the same trace. The scheduler has decided to move *Instruction A* from BB #1 into BB #2. While it is probable that the execution of BB #1 will be followed by BB #2 (otherwise they would not be in the same trace), it is also possible that the side exit to BB #4 will be taken. Then, unless a copy of *Instruction A* is placed along this path, it will not be executed when the side exit is taking, resulting in incorrect behavior. Nor can *Instruction A* simply be placed into BB #4. In that case, the execution path from BB #3 to BB #4 would be incorrect. Figure 7.2(b) illustrates the solution of traditional trace scheduling. By duplicating instruction A in a new basic block, BB #5, correct program behavior is maintained.

Similarly, Figure 7.2(c) describes the situation of an instruction that is moved up above a side entry. *Instruction A* cannot simply be copied into BB #3, because BB #3 may proceed to BB #4. The solution is to again create a new basic block for the compensation code, as shown in Figure 7.2(d).

The creation of new basic blocks for compensation code (BB #5 in Figures 7.2(b) and 7.2(d)) shows why compensation code can increase code size. This increase is particularly costly since often only one instruction can move to a compensation block, as in BB #5, yet an entire VLIW long-word would still be needed, on some architectures.

Cases where compensation code cannot be created Having considered the cases of moving an instruction 1) below a side exit and 2) above a side entry, we now consider the more difficult cases of moving an instruction 3) above a side exit and 4) below a side entry. Figure 7.3, illustrates the hazards of these remaining two cases. In Figure 7.3(a), an instruction is to move above a side exit. This is similar to Figure 7.2(a), where the instruction is to move *below* a side exit. The difference between these two cases is that in Figure 7.2(a), the instruction moves *out of* the execution path of the side exit, but in Figure 7.3(a), the instruction moves *into* the execution path of the side exit. Therefore, where as in Figure 7.2(b) a copy of the instruction must be reinserted into the execution path of the side exit, the situation of Figure 7.3(b) is that the instruction must be *undone* on the side exit. Figure 7.3(b) represents a much more difficult problem than Figure 7.2(b). For one thing, most instructions cannot be undone as easily as the add instruction. For another, even an add instruction might cause an overflow exception that would never have happened, had the instruction not been moved. The analogous situation for Figure 7.3(d) is equally difficult. For these reasons, no one, including us, proposes performing compensation code for the two cases in Figure 7.3.

There is a special instance in which the code motions of Figure 7.3 are allowed, however. This occurs when the compensation code is found to be unnecessary, through liveness analysis[59]. For instance, in Figure 7.3(b), the "r1" register may be dead in BB #3. Similarly, if we extend Figure 7.3(d) so that BB #1 has a side edge to a *new* block, BB #4, and if we find that the "r1" register is dead in BB #2, but not in BB #4, then Instruction A can move down, because it can skip placing the instruction into BB #2 entirely.

Even these special cases, however, must be restricted to instructions without side effects. Since this issue will become even more significant in our approach, we take a moment to describe it. Figure 7.4 illustrates how the divide and load operations may produce side effects. In Figure 7.4 (a), a divide instruction obviously

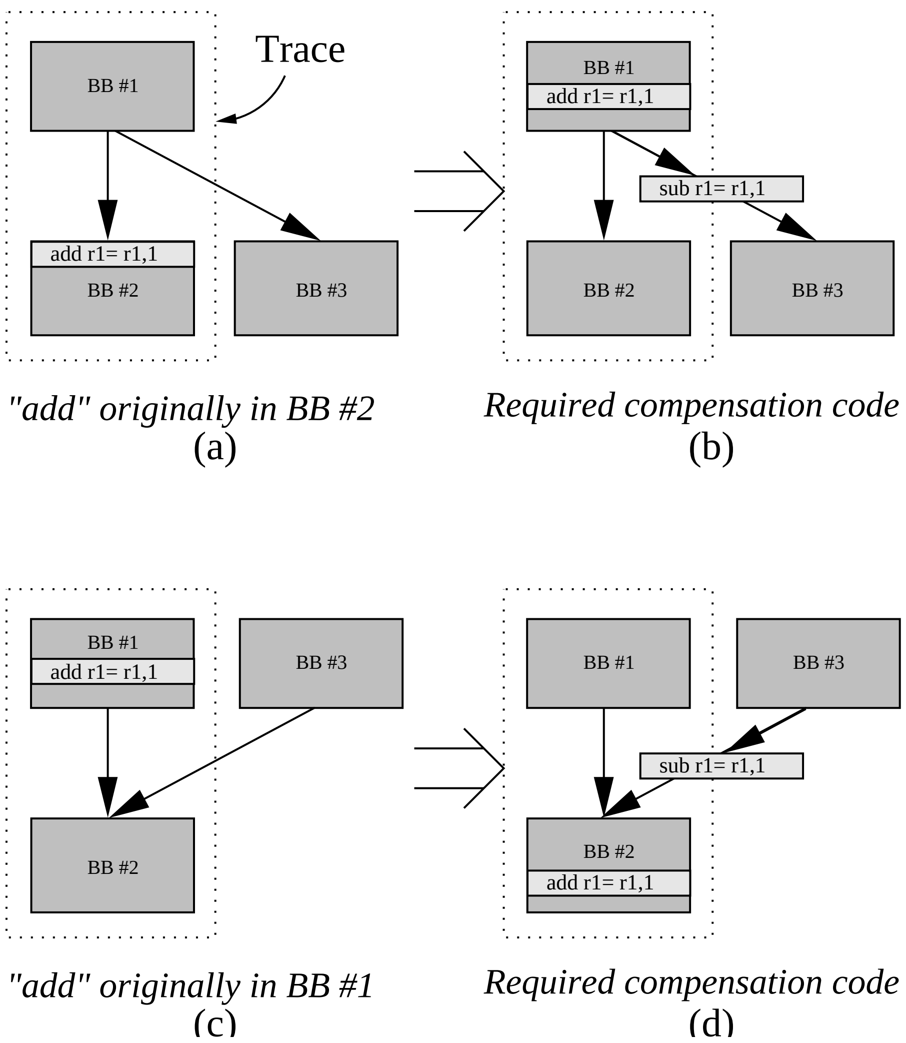


Figure 7.3: Examples of the theoretical compensation code that would be required to move an instruction above a side exit or below a side entry. The trace is composed of BB #1 and BB #2. BB #3 is a side block. In (a) an add instruction was originally in BB #2, below the side exit. In (b), the add has been moved into BB #1, and compensation code has been inserted along the off-trace edge to BB #3. Similarly, (c) depicts an add instruction that originally lies above a side entry, and (d) presents the compensation code that would be required. This figure is purely for conceptual understanding. In practice, such code motions are not allowed.

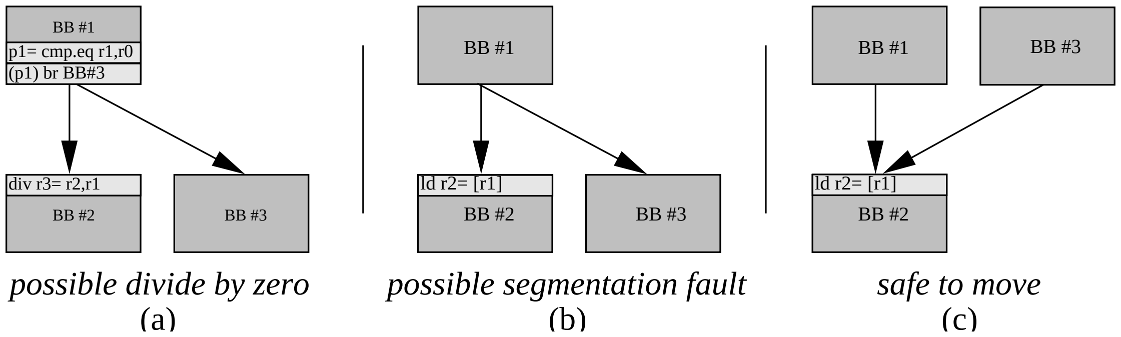


Figure 7.4: Examples of potential hazards in code motion. In (a) the division instruction cannot move above the test for zero. (The two lines of code shown in BB #1 are in IA-64 syntax. Together, they function as a branch-if-equal-to-zero.). In (b), moving the load instruction up would cause the load to execute even when the execution path is from BB #1 to BB #3. Even if the result is dead in BB #3, the compiler cannot guarantee that a segmentation fault that will not occurred. In (c), the load can be safely boosted up, because there are no paths that avoid the load.

cannot be moved above its divide-by-zero check. In Figure 7.4(b) a load instruction cannot move above a side exit, *even if the result register (r2) is dead* on the side exit to BB #3, because a load instruction may cause a segmentation fault. (To avoid this problem, some machines allow *speculative loads*, but we do not assume this property.) For clarity, Figure 7.4(c) emphasizes that these side effect restrictions only apply to the cases of Figure 7.3, and not those of Figure 7.2.

Reducing the amount of compensation code [3] proposes several optimizations to reduce the code size overhead of trace scheduling. The basic idea is to merge compensation blocks which share the same side edge. In addition, instructions which are dead do not need to be scheduled. For fairness of comparison, our trace scheduler performs a similar set of optimizations. In our implementation, of their scheme, the cost of compensation code is further reduced by allowing compensation blocks to be considered as a part of later traces, and by preventing an instruction from creating compensation code at more than one location in the trace. Figure 7.5 illustrates this concept. Figure 7.5(a) indicates that *Instruction A* is originally in BB #1. In Figure 7.5(b) *Instruction A* moves into BB #2, without the need of compensation on the edge to BB #6. Then in Figure 7.5(c) *Instruction A* moves into BB #3, creating a compensation block before BB #7. Next, in Figure 7.5(d) the instruction moves into BB #4. *Instruction A* cannot move into BB #5, because that would require additional compensation code before BB #9. A similar restriction is placed on upward code motion. We note that while our method does avoid the creation of many compensation copies when moving instructions across multiple branches, it does not prevent multiple copies at a single location in the trace, such as if two side entry edges pointed to the same block in the trace.

7.4 An across-block scheduling algorithm that decreases code size

The goal of our across-basic-block analysis is to constrain the movement of specific instructions across basic block boundaries, when those movements are likely to increase code size. In particular, we look for, code motions that will actually decrease the code size. In this way, it is possible to avoid the code-size cost (due to compensation) of trace scheduling, while at the same time enjoying the *code-size savings of across-block scheduling.*

The normal contribution of across-block methods such as trace scheduling and its variants is to provide a mechanism to move instructions during within-trace scheduling (a mechanism we borrow). The purpose of our across-block analysis is different: it runs *before* within-trace scheduling and *prevents* certain moves by

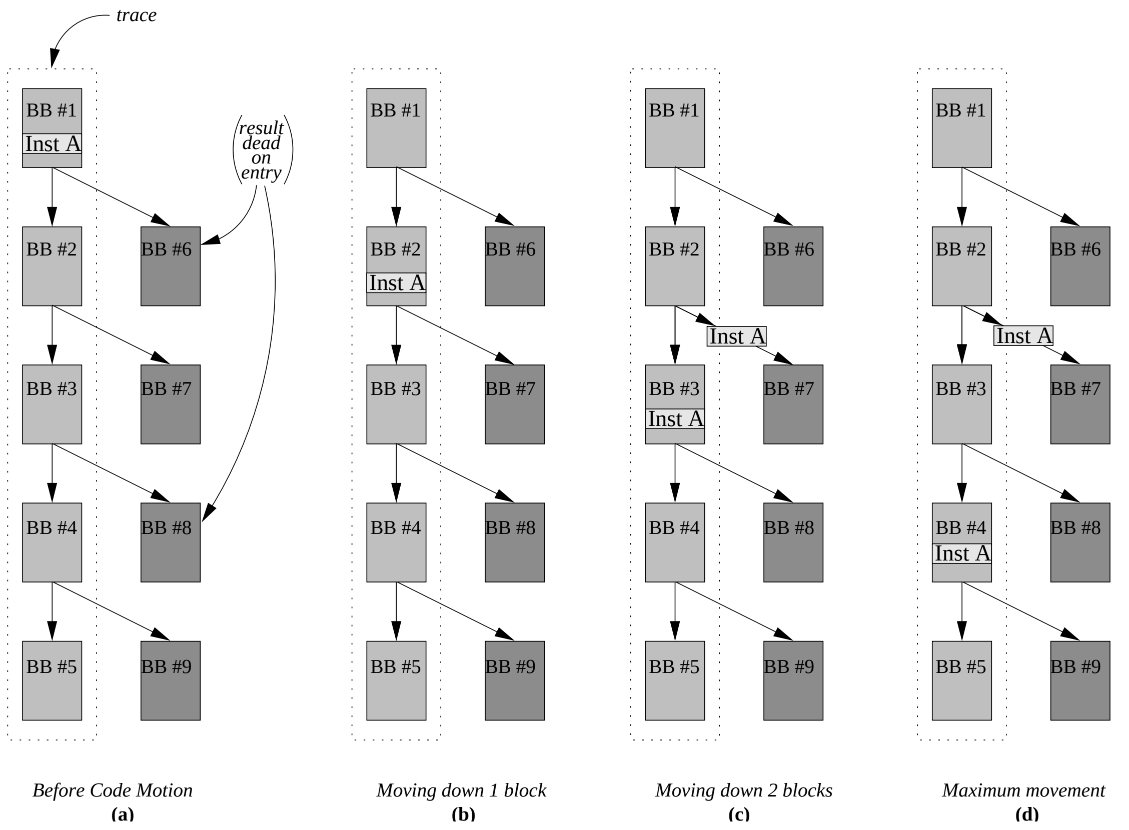


Figure 7.5: Illustration of reduced code duplication. In (a), *Instruction A* is in its initial basic block, BB #1. In (b) *Instruction A* moves into BB #2. In general, this would require a compensation block between BB #1, and BB #6, but in this case, *Instruction A* is found to be dead in BB #6, so no compensation is performed. In (c), *Instruction A* moves into BB #3, and creates compensation along the path from BB #2 to BB #7. In (d) *Instruction A* can move into BB #4 because it is dead into BB #8. *Instruction A* cannot move into BB #5, however, because this would require a second level of compensation code, between between BB #4 and BB #9. This approach reduces the amount of code size increase, without severely restriction the motion of instructions within traces.

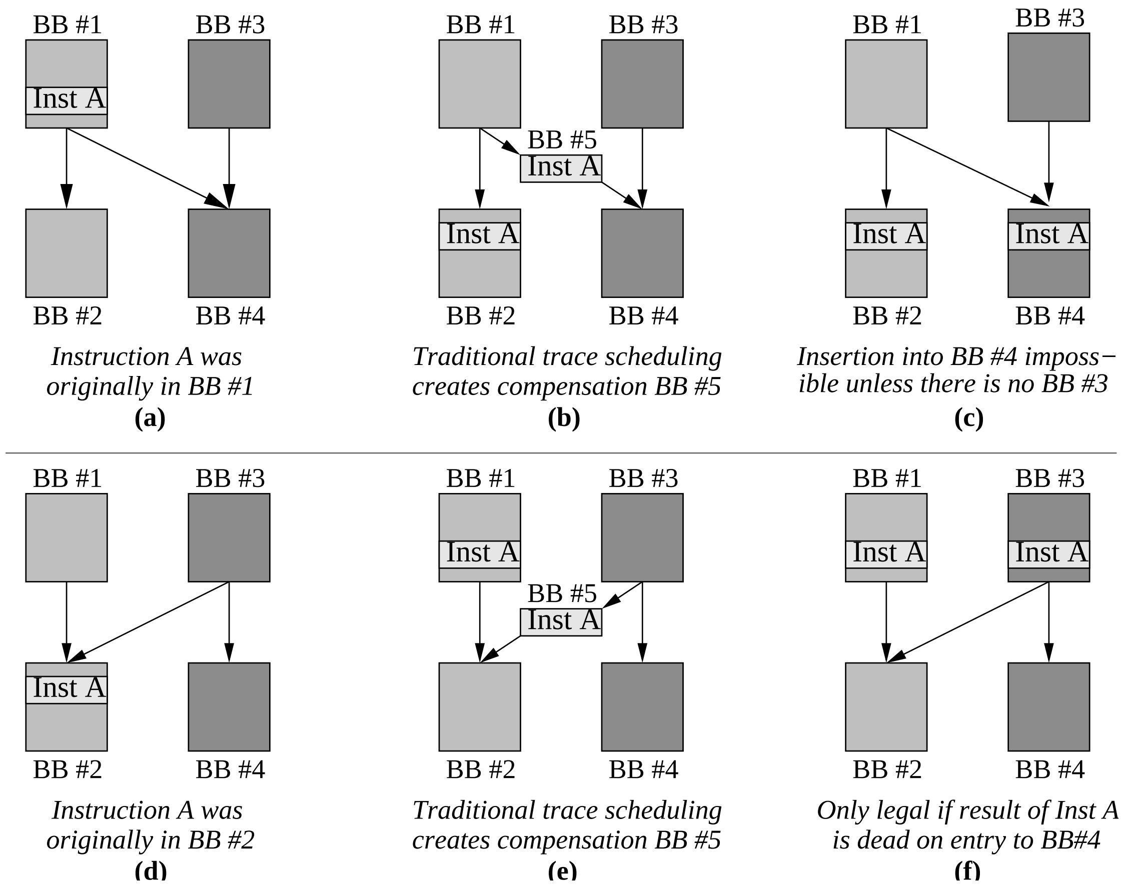

BB #1
BB #3
Inst A
BB #2
BB #4
Instruction A was originally in BB #1
(a)
BB #1
BB #3
BB #5
Inst A
Inst A
BB #2
BB #4
Traditional trace scheduling creates compensation BB #5
(b)
BB #1
BB #3
Inst A
Inst A
BB #2
BB #4
Insertion into BB #4 impossible unless there is no BB #3
(c)
BB #1
BB #3
Inst A
BB #2
BB #4
Instruction A was originally in BB #2
(d)
BB #1
BB #3
Inst A
BB #5
Inst A
BB #2
BB #4
Traditional trace scheduling creates compensation BB #5
(e)
BB #1
BB #3
Inst A
Inst A
BB #2
BB #4
Only legal if result of Inst A is dead on entry to BB#4
(f)

Figure 7.6: Scenarios for inserting compensation code. BB #1 and BB #2 are part of the trace. BB #3 and BB#4 are outside of the trace. In (a)-(c), *Instruction A* was originally in BB #1, but has been scheduled into BB #2. Compensation code is needed for the side exit to BB #4. In (d)-(f), *Instruction A* was originally in BB #2, but has been scheduled into BB #1. Compensation code is needed for the side entry from BB #3. (b) and (e) illustrate the solution of traditional trace scheduling. (c) and (f) illustrate our solution, which is not always feasible.

inserting appropriate dependence constraints. The rest of this section shows how compile-time analysis can detect if a movement will increase code size. We then give the details of our across-block scheduler.

How code size increase can be avoided Our method is to detect, through compiler analysis, those instances of code motion that do not increase code size, and to disallow all other instances of code motion. We now provide the insight into how compensation code may be accomplished without affecting code size.

There are three cases where compensation code does not increase code size. First, if the duplicated instruction is not needed in the off-trace path because its destination register is dead along that path. For example, in Figure 7.6(b), *Instruction A* can indeed be duplicated to both BB #2 and BB #4 if the destination register of *Instruction A* is dead upon entry into BB #4 - in fact it can skip BB #4 altogether. This case is easily discovered by the compiler. Second, if the instruction is moved *around* a side block, as shown in Figure 7.7. This case is also easily identified, by examining the control flow and register accesses of the side blocks. Third, if the duplicated instruction can fill a NOP in the off-trace block. For this to be the case, two conditions hold. (i) If, the off-trace duplicate instruction can move into an existing basic block, rather than creating a new block. For example, in Figure 7.6(b) instead of creating the new basic block BB #5, the duplicate copy of *Instruction A* can be placed directly into BB #4 *if* there no BB #3 (BB #4 has only one parent). (ii) And if, in addition, the off-trace duplicate replaces an existing NOP in the side block. For example, in Figure 7.6(b), moving *Instruction A* into BB #4 does not increase code size if it replaces an existing NOP within BB #4. Detecting when (i) is possible is easy since it depends only on the control-flow graph. However, detecting when (ii) is possible during scheduling is hard since, to know whether an instruction fits into a NOP, we need to have already finished scheduling the side block. In the next Section, we describe how we achieve this.

Paradoxically, allowing compensation code to increase the number of instructions, can have the effect of *reducing* the code size. Figure 7.8 illustrates this situation. *Instruction A* is moved from BB #1 to BB #2 and BB #3, fits in existing

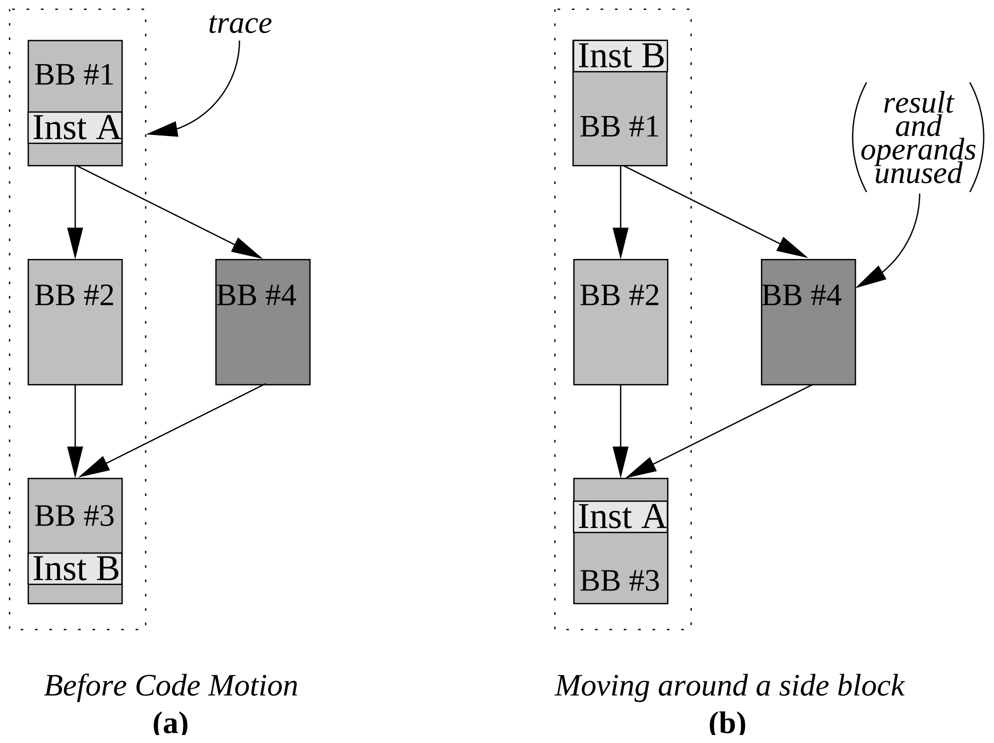


Figure 7.7: Avoiding compensation code when moving around a side block. BB #1, BB #2, and BB #3 comprise a trace. BB #4 is a side block whose only entry and exit edges lead back to the trace. In (a), *Instruction A* is originally in BB #1, and *Instruction B* is originally in BB #3. In (b), *Instruction A* moves to BB #3, and *Instruction B* moves to BB #1. These moments are only legal if neither the source nor destination registers of *Instructions A* and *B* are not modified between the instructions original position and its new one.

NOPs in both those blocks, and the entire long-word in BB #1 is eliminated, reducing the net code size by one long-word. Our algorithm needs no special case for handling such scenarios. Rather, the benefit is naturally obtained as a sub-case of the second case above, where our scheduler allows movement of the instruction since it does not increase code size. The fact that the code size is reduced is a bonus that is discovered by the within-trace scheduler later.

Our method detects all cases when code motion does not increase code motion as described above, and disallows all other moves, but it does not consider predication [40]. In Chapter 9, we will revisit this issue and find additional opportunities

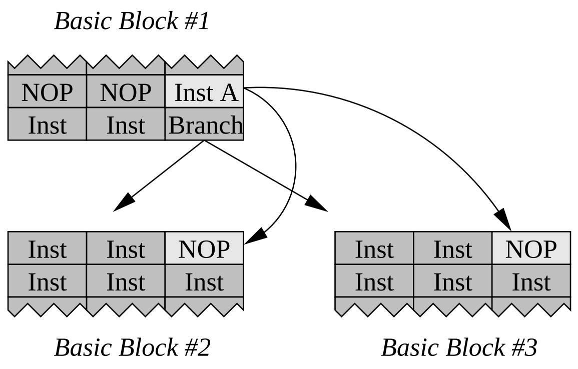


Figure 7.8: An example of how code size can be reduced even though an instruction is duplicated. Basic Blocks #1 and #2 are part of one trace, and Basic Block #3 is a side exit. If *Instruction A* is moved into Basic Block #2, it will fit into a NOP slot. The compensation copy placed into Basic Block #3 also fills a NOP slot. By moving *Instruction A* out of Basic Block #1, one of its long-words becomes empty and can be removed.

for code motion.

We now describe our algorithm for moving compensation code into NOPs. First, we determine the order in which to schedule traces. Second, we provide a heuristic for scheduling a trace when its side trace *cannot* be scheduled first. Third, we consider the difficulties of identifying whether compensation code can fill NOPs. Fourth, we describe our greedy algorithm for identifying when compensation code does, in fact, fill NOPs.

Scheduling order We begin by considering the order in which to schedule traces. The order of scheduling traces affects the results. For instance, in Figure 7.9, both *Trace A* and *Trace C* contain an instruction that can move into *Trace B*. In this figure, *Trace A* is far less frequently executed than *Trace C*. If *Trace A* were scheduled first, it could possibly fill the NOP slot of *Trace B*. This would, in turn, prevent code motion for *Trace C*. Since this constraint may impact the run-time of *Trace C*, and since *Trace C* is more frequent than *Trace A*, this ordering is sub-optimal.

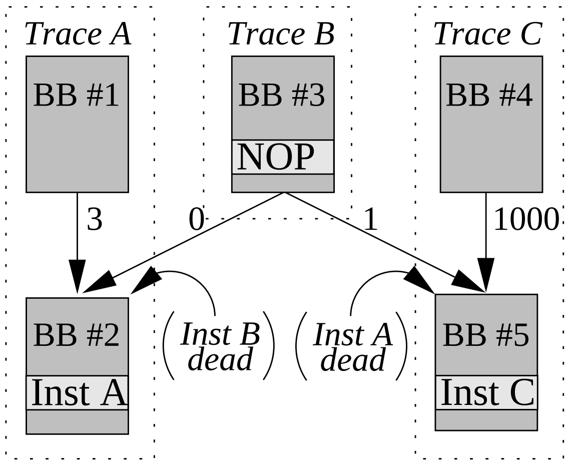


Figure 7.9: Contention for an available NOP slot. The control flow among three traces is shown. The edge weights indicate the frequency of the control edges. From these edge weights we see that *Trace C* is much more frequent than *Trace A*. Both of these traces have an instruction that can move up, and *Trace B* has a single available NOP slot for inserting an instruction. Although both instructions are able to fill the NOP, only one can be moved. Therefore, the order of scheduling *Traces A* and *C* will determine which instruction is allowed to move.

As a starting point for determining the proper scheduling order, we consider two cases where the solution is obvious. First of all, every trace that contains no potential for creating compensation code should be scheduled before other traces that do. One such example is a trace consisting of a single basic block. Another example is a trace where there are multiple blocks, but all instructions that can move are found to be dead on the side paths. By scheduling these simpler traces first, the other traces will have a greater likelihood that their blocks to insert compensation code into will have been already scheduled. The second case deals with the most frequent traces in the program. As will be explained in Chapter 8, these traces are allowed to move their instructions even when this increases the code size. Therefore, these traces can be scheduled with the assumption that their compensation code always fits in the side block. The details of identifying these traces are described in

Chapter 8; the point here is simply that, once identified, these traces can avoid the issue of determining when compensation code fits into the side blocks.

The remaining traces are scheduled in the order of their frequency, through an iterative process that we now describe. Once a trace, T, is chosen, it is possible that it may consider inserting compensation code into a side trace, ST, that has not yet been scheduled. If so, an *attempt* is made to schedule ST before scheduling T. Iteration may arise if the trace, ST also wants to put its own compensation code into its own side trace, SST. If SST is, itself, not yet scheduled, Then we must iterate.

A heuristic for when iteration fails This iterative procedure of scheduling side traces first is only an attempt, because two issues may arise that prevent its use. First, a cyclic edge may be encountered, where the side trace, SST, is in fact, an earlier trace in this iterative procedure. Second, the side trace may suffer from the contention problem indicated by Figure 7.9. In the context of this figure, *Trace A* would be one of the iteratively scheduled side traces of the current trace, and *Trace C* would be an unscheduled trace. We note that *Trace C* cannot be more frequent than the current trace, T, so this contention can only occur within the iterative process.

If either of the above problems prevents the iterative scheduling of a side trace, then we apply our heuristic, giving that side trace a *tentative schedule.* A tentative schedule is one where no code motion is allowed into those side blocks with contention or cycles. Tentative scheduling may prevent some opportunities for code motion within the current trace, but the effect is not overly significant. Tentative schedules also introduce further complexity into the scheduling algorithm. As the traces are scheduled, in the order of their frequency, the current trace may now be in one of three states: 1) not scheduled, 2) scheduled (either by virtue of being one of the two easy cases, or due to having been iteratively scheduled as the side block of an earlier trace), and 3) *tentatively* scheduled. If the current trace has been tentatively scheduled, it will now be rescheduled. However, the tentative solution serves as an improved lower bound to the branch-and-bound, within-trace scheduler. In addition, tentative blocks may be encountered during the iterative process of scheduling side blocks. In these cases, it is not desirable to reschedule the tentative block, *unless* it no longer needs to be tentative. In this way, any particular trace will never be scheduled more than twice, and will only be scheduled twice in instance where this is needed for providing flexibility to more frequent traces.

Difficulties in seeing if compensation code fits into the side trace Once all side traces have been scheduled (at least tentatively), the current trace may proceed. Since the within-trace scheduler employs an exhaustive search, it will continually experiment with moving the instructions across the basic block boundaries, *even if it doesn't end up deciding to move them.* As a result, the within-trace scheduler will constantly test whether a given instruction can fit its compensation code into a NOP on the side trace. Since this test will be performed repeatedly, *inside the recursive loop* of our within-trace scheduler, light-weight heuristics must be found to provide conservative answers for whether instructions can fit into NOP slots.

There are three primary causes for the difficulties in determining whether instructions from a single trace can fit into the NOPs of a side trace. First, individual instructions may be movable, and yet may not be movable together. For instance, if two instructions wish to move into the same side block, and if that side block has only a single NOP slot, then either instruction can move, but not both. In fact, the code motions can be quite complex, and instructions from different blocks may move into the same side block. Figure 7.10 illustrates this situation. This figure describes a potential scenario resulting from our compensation-code algorithm. In this figure, Basic Blocks #1, #2, #3, and #4 form the trace. Originally, Instruction A is in Basic Block #1, Instruction B is in Basic Block #2, and Instruction C is in Basic Block #4, as shown in Figure 7.10(a). After code motion, all three of these instructions might be moved into Basic Block #3, as shown in Figure 7.10(b). Compensation copies of these instructions would then be inserted into Basic Block #5, as also shown. From Basic Block #5 of Figure 7.10(b), we see two important points related to finding available slots for compensation. First, a single off-trace block may have compensation copies inserted from several different within-trace blocks. Second, these copies may be scheduled in any order (in this case, *Instruction A* is scheduled after *B* and *C*).

A second complication is that the specific scheduling choice for the side block may affect which instructions may move. For instance, an instruction cannot move down into a side block for which it has a data dependency to an instruction in the first long-word of the side block. The dependent instruction might not, however, need to have been scheduled that early. For another instance, in machines with

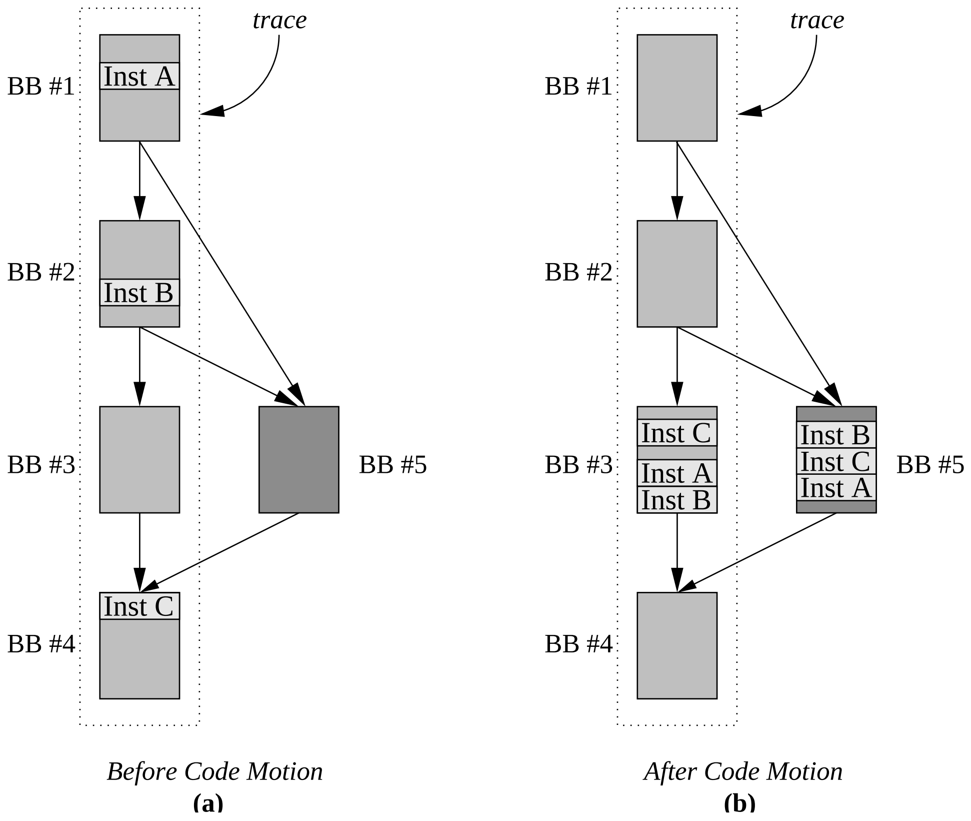


Figure 7.10: An example of how multiple on-trace blocks may insert compensation code into the same off-trace block.

templates or other asymmetries between VLIW slots, the NOP to be filled may not match with the type of the instruction to move. It is very likely, however, that the long-word could be rearranged so that the allowed types of the NOP would permit the compensation instruction to fit.

A third problem is that instruction we wish to insert into the side block may have data dependencies. For instance, a compensation instruction may be place in a side exit block, and it might write to a register than an instruction in the side block later reads from. Therefore, compensation code cannot simply schedule anywhere in the side block, but must obey data dependencies. A more complex issue is dependencies with other compensation code instructions.

A greedy algorithm for ensuring that compensation code fits into the side trace In summary, we clearly need a fast heuristic, that is still able to cope with the two scheduling difficulties just mentioned. We now describe our a greedy scheduler that accomplishes this. The process has six steps. First, even before calling the within trace scheduler, every instruction that might possibly require compensation are all temporarily place into side blocks, and the data dependencies to each other and to the instructions in the side trace are observed, before removing them. Second, since the side blocks have been scheduled already, Each of its instructions have known schedules. We therefore can use the data dependencies of the compensation instructions to identify the latest cycle (downward motion) or earliest cycle on which they can schedule into the side block. In this way, data dependence analysis does not need to be performed inside the recursive search of within-trace scheduling. Third, once a trace is being scheduled, and we need to test whether an instruction fits in the side trace, we attempt to insert it in each of the available cycles on the side block, starting with the deepest allowed cycle within the side block. Fourth, for each cycle that we attempt an insertion, we employ [1]'s quick FSA to permit a possible re-assignment of the instructions of that cycle to the available slots. This is superior to keeping the current slot assignment, but inferior to allowing the instructions in the side trace to move to different blocks. Fifth, if the scheduler later decides not to move the instruction after all, it removes it. Sixth, after scheduling, some instructions may have been moved when it was not necessary. Therefore, we apply the same greedy approach to pushing the instructions back into their original blocks. We note that this never effects the run-time, and may free up NOPs for later traces.

In our results, we find that this approach allows us to perform most of the relevant code motion as trace scheduling.

Chapter 8

Doing better by considering extremes

Up until now, our approach attempts to optimize for both run-time and code size. In Chapters 4-6, we find the schedule with the smallest code size among those with minimum run-time. In Chapter 7, we use compensation code to reduce the run-time, but only when it does not increase code size. In this section we observe that, for basic blocks at the extreme ends of the spectrum, we can do better by removing one of the constraints, either for code size or for run-time. For those traces that are most frequent, it is reasonable to optimize for run-time only, even at the cost of code size. In a complementary way, those traces that are most *in*frequent should be optimized for code size only, even at the cost of run-time. Figure 8.1 pictorially depicts these observations. Such a trade-off between two objectives is new since it is not needed for existing methods which optimize for run-time only in all cases.

Frequent traces For the most frequent traces, run-time becomes more important than code size. To adapt our methods for a goal of run-time only, the within-trace methods of Chapters 4-6 do not require modification, as they already search for a minimum run-time solution. The across-block analysis of Chapter 7 could negatively affect run-time, however, because it restricts code motion. Therefore, for frequent traces, we do not apply the across-block movement constraints of Chapter 7. The frequent traces are defined as those that are executed more often than

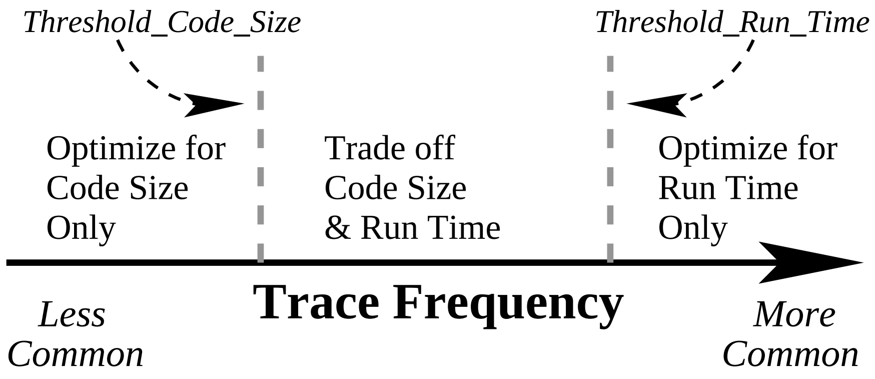

Threshold_Code_Size
Threshold_Run_Time
Optimize for Code Size Only
Trade off Code Size & Run Time
Optimize for Run Time Only
Trace Frequency
Less Common
More Common

Figure 8.1: Scheduling strategies along the trace frequency spectrum. Traces are sorted by frequency. Those with a frequency below an experimentally determined threshold, *Threshold_Code_Size*, will be scheduled with minimal code size. For those traces whose frequency exceeds *Threshold_Run_Time*, only run-time will be targeted.

an experimentally defined threshold, *Threshold_Run_Time*. This threshold value is found by repeating the method with different thresholds and choosing the best value, as is done in the results in Chapter 10. Based on the general rule that most of a program's execution time is spent in a small portion of the code, we choose *Threshold_Run_Time* so that the sums of the frequencies of all of the blocks larger this threshold adds up to the desired percentage of the total execution time.

Infrequent traces For the least frequent traces, code size is the over-riding concern. This requires a modification to the within-trace methods of Chapters 4-6, but not to the across-block restrictions in Chapter 7. Our within-trace methods are modified to optimize for code size alone by simply setting all of the data dependence latencies to a value of 1 in the Data Flow Graph. This is because, when the dependence latencies are all 1, the minimum-run-time solution *is the same* as the minimum-code-size solution, since both are directly proportional to the number of long-words. The resulting solution is still correct for the original latencies by inserting appropriate stalls, and the code size is optimal since the minimum code size solution does not depend on the instruction latencies.

A second optimization performed for infrequent traces is register reallocation. Since the pro64 compiler's pre-existing register allocator is targeted for run-time, infrequent traces can benefit from a reallocation that reduces register reuse. Register reuse improves run-time, because using fewer registers results in few register spills. But register reuse also creates anti-dependencies and output-dependencies, because reusing a register erases its old value, so that all reads or writes of the old value must be required to precede all writes of the new value. For infrequent traces where run-time is not important, register reuse becomes undesirable. Therefore, we perform a simple reallocation for these traces, converting reused registers into separate register locations, whenever it is legal and beneficial.

We now describes our register reallocation strategy. Reallocation candidates are identified based on the anti and output dependencies in the DFG. Yet these registers can only be renamed if two conditions hold. First, the renamed register cannot be used in code regions that are not infrequent, because this would increase register spills. Since register usage is allocated at the beginnings of functions in our test machine, this means that reallocation should only be performed for functions that are completely infrequent. Second, the register must be truly reused for *different* variables. For instance, consider the absolute value function, shown in Figure 8.2. This figure contains an anti-dependency edge, because the compare instruction reads register r32, and then the subtract instruction writes to it. Yet this anti-dependence does not indicate a re-namable register, as Figure 8.2(b) demonstrates; if r32 were to be renamed in the second instruction, then the register to use for the third instruction becomes ambiguous. This observation is simply the result of the fact that this anti-dependency edge does not indicate register reuse, but rather the updating of a live variable. To avoid incorrectly renaming in such cases, one must verify that no instruction reading from a particular register has reaching definitions that will be given different names. For ease of implementation and for faster compilation, however, we use a more restrictive pair of requirements: 1) the overwriting operation (*i.e.,* the operation pointed to by the tail of the anti-or-output dependency) must not be predicated and 2) the basic block containing the overwriting operation must dominate all of its descendant blocks for which the reused register is not dead.

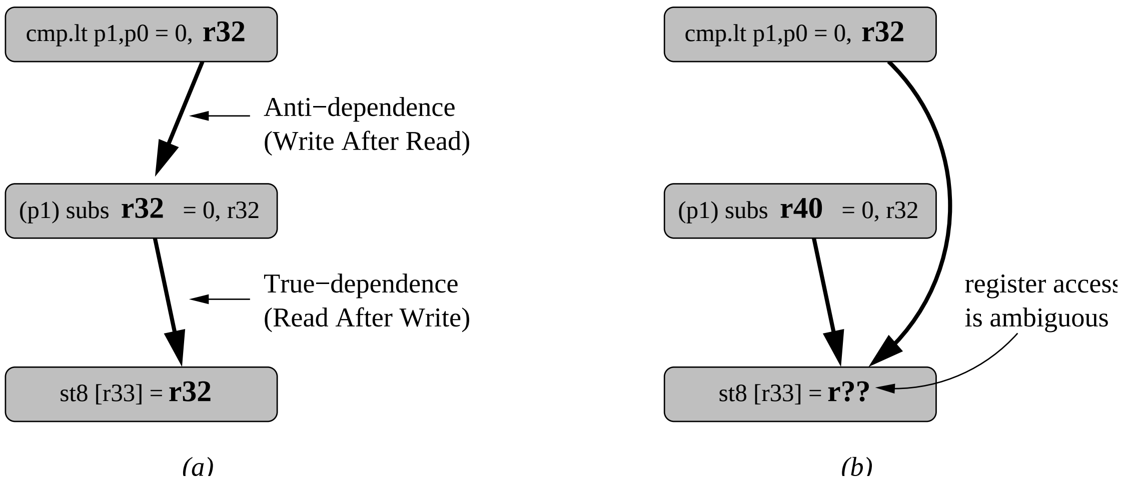


Figure 8.2: The DFG for a possible use of the absolute value function, showing an anti-dependency that does not indicate a re-namable register. (a) indicates an an anti-dependency between the first instruction and the second. (b) illustrates how this anti-dependence does not correspond to a re-namable register. Since the second instruction does not always execute, the correct register for the third instruction is ambiguous, which is to say that “r32” is being used as a single variable, despite the anti-dependency

Chapter 9

Increasing code motion opportunities through predication

Predication is becoming more common in embedded VLIWs [56, 18]. When predication hardware is present, it allows our algorithm greater flexibility in moving instructions across basic blocks. In fact, we can then move code with nearly as much freedom as in traditional trace scheduling, but without creating compensation blocks.

In achieving this goal, we have developed both a new way of looking at predicates and a new predicate analysis tool. These contributions are novel and applicable to a variety of problems, in addition to the use that we have made of this analysis.

In this chapter, Section 9.0.1 introduces the concept of predication. Section 9.0.2 explores prior work on using predication for instruction scheduling. Section 9.0.3 describes our new predication analysis. And Section 9.0.4 presents our modified algorithm for when predication is present.

9.0.1 Predication

Predication is a hardware mechanism for converting control dependencies into data dependencies. A special *predicate bit* is associated with every instruction that is executed; if the corresponding predicate bit evaluates to false, then the instruction will not be executed (*i.e.,* . the instruction will be converted into a NOP). Thus, the predicate bit functions as an additional operand to the instruction. Because each instruction is assigned its own predicate, predication permits the conditional execution of instructions within a basic block.

The motivation behind predication is that the basic blocks of real programs tend to be too small to exploit the parallelism available in modern machines. This is particularly a problem for VLIWs, since the long-words are scheduled at compile time, and only instructions from a single basic block may be safely scheduled on the same cycle. Predication, however, allows a group of separate basic blocks to become a single large block. This larger block is likely to have more scheduling flexibility, and to therefore allow more parallelism. In effect, predication allows for the VLIW what branch prediction and out-of-order-execution hardware allow for the superscalar: parallel, speculative execution of instructions from different basic blocks.

Figure 9.1 presents a simple use of predication. In this figure, the abs() function is implemented using both the traditional approach and predication. In Figure 9.1(a), positive values will result in branching over the negation instruction. In Figure 9.1(b), the negation instruction is always issued, but is only executed when its predicate indicates a non-positive value. From Figure 9.1, we can see that predication has converted control flow into data flow, resulting in one basic block where there were originally three.

Figure 9.1(b) is also useful for demonstrating the syntax of predication, employed by the IA-64. The predicate bit is indicated by the parenthesis to the left of the opcode. The sub instruction only executes when p2 is ‘1’, but the cmp instruc-

```
   bgt  r1,r0,L1;; if r1>0 goto L1     (p0) cmp.gt.un p1,p2 = r1,r0
   sub  r1= r0,r1;;r1 = 0-r1           (p2) sub r1= r0,r1
L1:
                (a)                                      (b)
```

Figure 9.1: Two implementations of the absolute value function: Traditional (a) and Predicated (b). In these examples, as in many real machines, r0=0, and p0=1.

tion always executes, because p0 is hardwired to '1'. Therefore, the cmp instruction is not really predicated at all; in the future the p0 will not be shown, but simply assumed for non-predicated instructions. The cmp opcode has two extensions: 'gt' and 'un'. The 'gt' indicates that the comparison will test for whether r1 is **g**reater **t**han 0. The meaning of 'un' is that it is a normal compare operation. We will soon see that there are other predication modes, but these are not needed at the moment. The cmp instruction also illustrates how predicates are assigned. A comparison always writes to two predicates, because the second is assigned the compliment of the first. In this figure, the cmp instruction sets p1 to true if and only if $r1 < 0$, and it places the compliment of p1 into p2.

The 'un' extension that we see in Figure 9.1(b) is one of eight operation modes of the compare opcode which we will now explain. The others are shown in Table 9.1. Table 9.2 describes the behavior of each. In Table 9.2, a '-' indicates that the predicates are left unchanged. From this figure, we see that only the unconditional cases ('un' and 'uc') write a result even when the input predicate is '0'. Otherwise, the unconditional and conditional modes are identical. The Parallel Or and Parallel And forms have special uses described in Section 9.0.2. For our purposes, it is sufficient to assume that the second predicate will be written in the same instances as is the first predicate, and with complementary values. Occasionally, more complex

extension	Meaning of Abbreviation
un	Unconditional, Normal
uc	Unconditional, Complemented
cn	Conditional, Normal
cc	Conditional, Complemented
on	Parallel Or, Normal
oc	Parallel Or, Complemented
an	Parallel And, Normal
ac	Parallel And, Complemented

Table 9.1: Possible compare extensions.

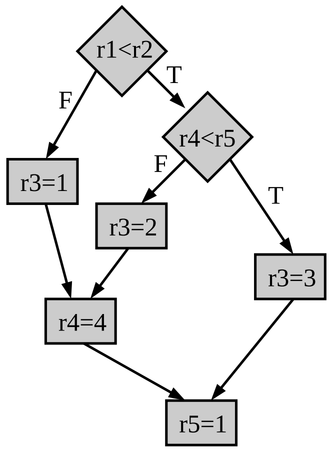
predication modes are provided, as in IA64. These special modes are not significant to the problem addressed here.

input predicate	result of compare	value written to first predicate							
		un	uc	cn	cc	on	oc	an	ac
0	0	0	0	-	-	-	-	-	-
0	1	0	0	-	-	-	-	-	-
1	0	0	1	0	1	-	1	0	-
1	1	1	0	1	0	1	-	-	0

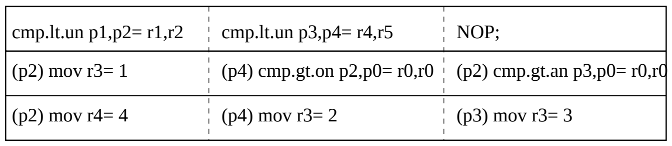
Table 9.2: Behavior of compare operations.

9.0.2 Related work in predication

The first use of predication was for *if conversion* [40, 41, 42], a procedure that merges the if and else portions of code into a single basic block. The advantage of if-conversion is that it allows for larger basic blocks. The disadvantage is that, when the *if* and *else* blocks are large, a high number of instructions will be dropped, effectively reducing the width of the VLIW, and harming performance. Therefore, heuristics are used to ensure that the if-conversion is beneficial. In addition, if-conversion is applicable to more than simple if-else blocks. For instance, Figure 9.2 gives an example of a more complicated region with three control flow paths. In this



(a)



cmp.lt.un p1,p2= r1,r2	cmp.lt.un p3,p4= r4,r5	NOP;
(p2) mov r3= 1	(p4) cmp.gt.on p2,p0= r0,r0	(p2) cmp.gt.an p3,p0= r0,r0
(p2) mov r4= 4	(p4) mov r3= 2	(p3) mov r3= 3

(b)

Figure 9.2: Using predication in larger regions. (a) is a flow diagram for a particular region of code. This diagram corresponds to 7 basic blocks. (b) is the corresponding predicated code on a 3-wide VLIW.

example, all seven basic blocks have been converted into a single basic block, which may execute in 3 cycles on a 3-wide VLIW. Examining the code in Figure 9.2(b), we see the usage of the "on" (OR) and "an" (AND) compare operators from Table 9.2. Also, in this figure, the comparison $r0 > r0$ is made, because this evaluates to false and is used because Boolean manipulation of predicates is not supported ([58] have proposed introducing instructions for just such Boolean manipulation.) If-conversion may be employed within the framework of our analysis.

Predication has also been used to assist software pipelining. Software pipelining is a method of exploiting parallelism across loop iterations. Unlike loop unrolling, which requires large code-size overhead, Software pipelining does not unroll the loop. In the presence of predication, the software pipelining approach becomes much more elegant, because no epilogue or prologue code is needed. This use of predication is not relevant to our algorithm.

One form of predication analysis that has been proposed is in [43]. This work observes that predication interferes with traditional dataflow analysis. For instance, the *reaching definitions* to a register must now consider that a predicated write may not necessarily kill the previous definition. [43] extends dataflow analysis to handle predication, and uses this new analysis to overload resources. The idea is that, if two instructions are predicated on mutually exclusive predicates (a common occurrence in if-conversion), then they may write to the same register, without any data dependencies between them. In [44] a hardware modification is proposed to further extend this idea to allow instructions to share the ultimate resource - their VLIW slot. A VLIW is imagined, where the long-words may be longer than the VLIW width, provided that the compiler can guarantee that the true width fits in the VLIW. [44] extends this idea to allow overlap of probabilistically independent instructions.

In all of this work, predication analysis is limited to single trace regions; the only prior work to consider global predicates is [58]. [58] proposes a new predication analysis scheme. It transforms program control logic into a program decision logic network, then employs Boolean simplification techniques to reduce the complexity of control flow. This approach is beneficial to programs with very small basic blocks. The type of analysis performed here is unrelated to our new analysis.

In contrast to our analysis, most existing predication analysis is concerned with identifying mutually exclusive predicates, rather than with finding their actual values. This difference arises from the way in which existing research uses predicates.

9.0.3 A new predication analysis

Our approach to predicate analysis for the first time identifies the actual values of many predicates at any point in the code. The analysis is a form of reaching definitions analysis, with two unique properties. First, the possible values are Boolean. Second, the definitions of these variables occur on the *control flow edges*, rather than in the basic blocks.

Figure 9.3 illustrates our data flow analysis. In Figure 9.3(a), BB #1 writes to p1 and p2. We do not know the result of this comparison however. Existing approaches contend themselves with observing that p1 and p2 are mutually exclusive, or with attempting to reason about the behavior of r1 and r2. We note, however, that the values of both p1 and p2 become known, once the "(p2) br BB #2" instruction is executed. On the control edge between BB #1 and BB #2, we can deduce that p2 is True, and hence p1 is False. We further note in Figure 9.3(a) that BB #2 writes to predicates p2 and p3. Hence, at the end of BB #2 the value of p2 is no longer known, since it does not end with a branch. Importantly, p1 is still known to be False on exit from BB #2. Even if the value of p1 is dead, its value is still *known*, and can be used. Figure 9.3(b) summarizes what we have learned. This graph does not lend itself to dataflow analysis, because the kills occur inside of blocks, but the generates occur *on control edges*. In Figure 9.3(c) we split all blocks with two successors into two blocks, so that the generate and kill information may be consolidated. In this figure, separate generates and kills are created for True predicates and for False predicates. This is logical, since any particular predicate

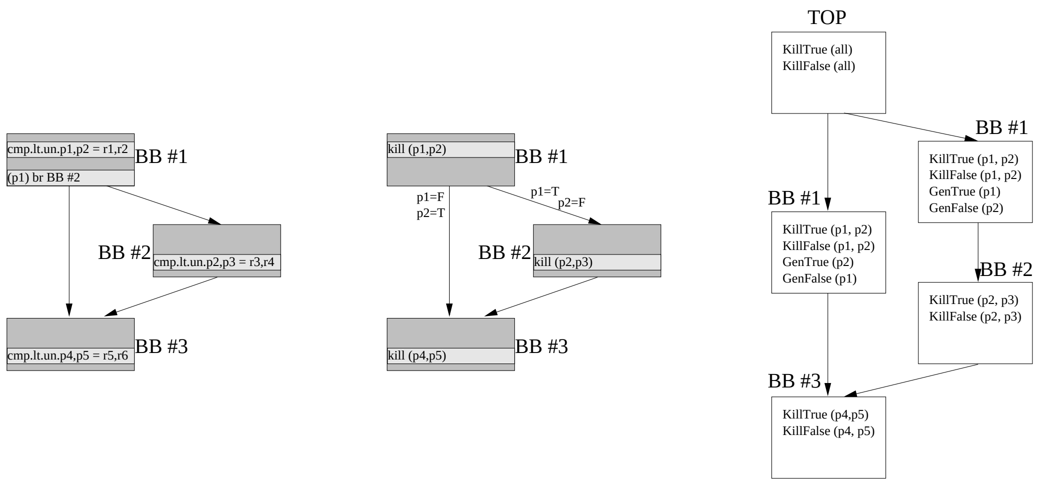


Figure 9.3: Converting predication into a dataflow problem. (a) illustrates a simple function with three basic blocks. It also indicates the instructions that write and use predicates. (b) condenses the information from the instructions into simple observations of where predicate values become known and unknown. (c) converts the the Control Flow Graph into a data flow graph. In (c), BB #1 has been split into two nodes. This is because each edge out of BB #1 has different Gen sets. We also note that two separate data flow analysis are represented simultaneously - one for True predicates, and one for False.

may be: 1) known to be True, 2) known to be False, or 3) unknown.

Predicates can also be renamed to improve this analysis in two ways. First, we note that join points may unnecessarily nullify predicates. For example, in Figure 9.3, the value of p1 is True on the edge from BB #1 to BB #3, and False on the edge from BB #2 to BB #3. Therefore, it becomes unknown in BB #3. In this case, nothing can be done about this, but sometimes, renaming one of the predicates can preserve information. Renaming has no cost, because it simply involves changing the predicate opcode. Second, renaming allows us to use this analysis to find a predicate with specific properties of being True on certain control edges and False on all others. For instance, consider if a predicate, p5, has the right behavior on all edges but one. On that edge, it needs to be False, but is not. If another predicate, p7, *is* known to be False on that edge, then we can rename p7 to p5, so long as p5 is dead. Another method to accomplish the same purpose would be to insert a new, predicate writing instructions to force p5 to False along that path. Such an instruction might even be able to fill a NOP slots, and therefore not increase code size. We do not propose doing so, however.

In the next section, we discuss how we can use this new analysis to increase opportunities for code motion. We note, however, that this new analysis may have additional benefits to other problems of instruction scheduling. Three such problems are: 1) for providing additional information to determine mutual exclusivity properties for resource overloading, 2) improving code motion opportunities for traditional trace scheduling even in environments where code size does not matter, and 3) for safely boosting loads so as to avoid the load's miss penalty.

9.0.4 Our algorithm with predication analysis

For machines that employ predication code motion becomes possible in cases where the algorithm of Chapter 7 does not allow it. To understand how, we must begin by reviewing the restrictions imposed by Chapter 7. With the exception of the most frequently executed blocks (as defined in Chapter 8, compensation code is inserted directly into the existing side block, rather than into a newly-created compensation block, as traditional trace scheduling would do. Figures 9.4 is reproduced here from Figure 7.2 for ease of discussion. Figure 9.4(c) and (f) illustrate the conditions under which our earlier algorithm might prevents code motion, if the instruction is live outside the trace. With predication however, the instruction can

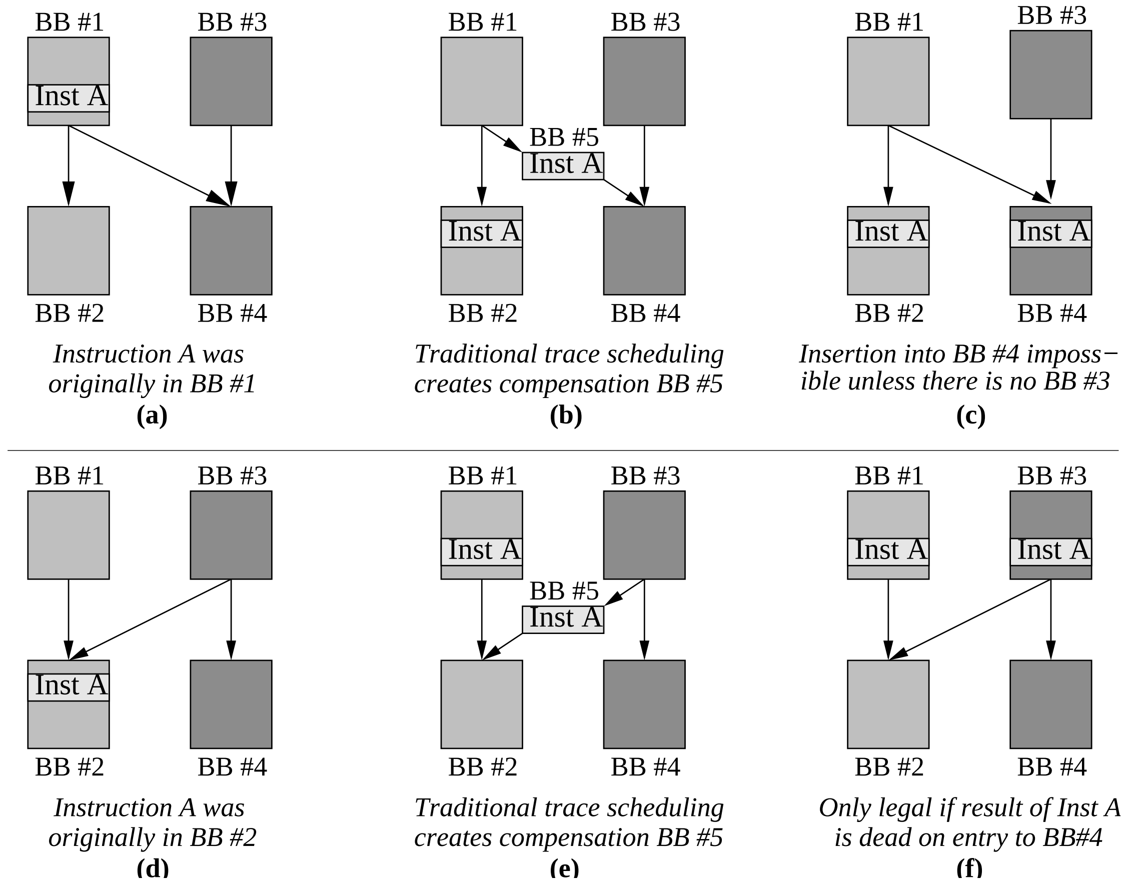

BB #1
BB #3
Inst A
BB #2
BB #4
Instruction A was originally in BB #1
(a)
BB #1
BB #3
BB #5
Inst A
Inst A
BB #2
BB #4
Traditional trace scheduling creates compensation BB #5
(b)
BB #1
BB #3
Inst A
Inst A
BB #2
BB #4
Insertion into BB #4 impossible unless there is no BB #3
(c)
BB #1
BB #3
Inst A
BB #2
BB #4
Instruction A was originally in BB #2
(d)
BB #1
BB #3
Inst A
BB #5
Inst A
BB #2
BB #4
Traditional trace scheduling creates compensation BB #5
(e)
BB #1
BB #3
Inst A
Inst A
BB #2
BB #4
Only legal if result of Inst A is dead on entry to BB#4
(f)

Figure 9.4: A reproduction of Figure 7.2, to illustrate the restrictions imposed by Chapter 7. Here, BB #1 and BB #2 are part of the same trace. BB #3 and BB#4 are outside of the trace. In (a)-(c), *Instruction A* was originally in BB #1, but has been scheduled into BB #2. Compensation code is needed for the side exit to BB #4. In (d)-(f), *Instruction A* was originally in BB #2, but has been scheduled into BB #1. Compensation code is needed for the side entry from BB #3. (b) and (e) illustrate the solution of traditional trace scheduling. (c) and (f) illustrate the insertion of the compensation code directly into the side block.

always be moved into the side block, provided that a predicate can be found with the property of only being true on the path from (Figure 9.4(c)) or to (Figure 9.4(f)) the trace. Predication also allows instructions to move above side exits and below side entries, something that even unconstrained trace scheduling could not usually allow (See Chapter7).

Figure 7.2(a) shows an example of when compensation is needed. In this figure, BB #1 and #2 are part of the same trace. The scheduler has decided to move *Instruction A* from BB #1 into BB #2. While it is probable that the execution of BB #1 will be followed by BB #2 (otherwise they would not be in the same trace), it is also possible that the side exit to BB #4 will be taken. Then, unless a copy of *Instruction A* is placed along this path, it will not be executed when the side exit is taking, resulting in incorrect behavior. Nor can *Instruction A* simply be placed into BB #4. In that case, the execution path from BB #3 to BB #4 would be incorrect. Figure 7.2(b) illustrates the solution of tradition trace scheduling. By duplicating instruction A in a new basic block, BB #5, correct program behavior is maintained.

When predication is allowed, however, this restriction can often be removed. Predication allows instructions to be inserted into side blocks even when the result register is live, provided that a predicate with the appropriate behavior can be found or constructed.

The construction of a predicate with the desired properties is an interesting problem. For instance, in moving an instruction below a side entry as in Figure 9.4(c), it is necessary to find a predicate which is true from BB #1 to BB #4, but not true from BB #3 to BB#4. Predicate analysis can identify such a predicate. However, predicate analysis might also tell us that, while no such predicate exists with the desired property, but renaming can create one. For instance if p1 is known to be true from BB#1 to BB #4, and if p2 is known to be false from BB #3 to BB #4, and if p1 is dead in BB #3, then p2 may be renamed to p1 in BB #3, or its ancestors. This renaming must avoid conflicts when renaming multiple predicates.

In our algorithm, predication is a 3 step process. First, predicate analysis identifies predicatable-movement candidates. Second, Our scheduler identifies the instructions that are moved. Third, if any predicate move candidates did in fact move, and if the predicate used requires renaming, then the renaming occurs.

Chapter 10

Results

Evaluation environment and benchmarks Because of compiler source code availability and access to a physical processor, we chose our evaluation platform to be a hypothetical 6-wide embedded VLIW whose instruction set is identical to the Itanium [6, 13], an Intel desktop VLIW. The variable-width long-words feature of the Itanium is not allowed, however, so as to represent a more typical VLIW processor for embedded systems. The speculative load and hyperblocking features are also not permitted, on similar grounds. A 6-wide VLIW is not excessive for embedded systems; for example, the TI C6000 series uses a width of 8. Although we model a fixed-width VLIW, the compiled code is still compatible with the IA-64 ISA, and so can be evaluated on a real Itanium.

An IA-64 instruction set was chosen since the most sophisticated open-source VLIW compiler we could find was for the Itanium instruction set. Using a sophisticated VLIW compiler has two advantages: first, it reduced the implementation effort since much functionality already existed, and second, by using a mature VLIW compiler, we could have confidence that our scheme improves performance versus the best available schemes. Our algorithm is implemented by modifying the Pro64 (v 0.11) research compiler for Itanium, run with full optimization.

Some of the important parameters of our test machine are as follows. It has two memory units, two integer units, two floating point units, and three branch units. Each of these unit types is asymmetric. For instance, some M-type instructions are only executable on one of the memory units, while others are executable on both. There is also a super-type instruction class, A, that can execute on either an I or M unit, but which has different latencies depending on the unit that it executes on. It also may place ordering restrictions upon the instructions that are run in parallel. All of these features increase the difficulty of the compiler's task, and thus challenge our method more. Our approach is equally applicable to less constrained systems.

Experiments are run on ten benchmarks from the MiBench suite of embedded applications. All are written in C. The benchmarks are described in Table 10.1.

Comparison Algorithm Our algorithm is compared to an implementation of the FSA-based across-block scheduler in [1] by Bala et. al., which is augmented by the technique in [3] by Freudenberger et. al. that aims to reduce the code size of compensation code. The reason for this choice is that we aim to compare against one of the best across-block schedulers among those aiming for run-time reduction ([1]), combined with one of the only techniques for reducing code size for such an approach([3]). There are better-for-run-time schedulers than [1], but most of them, such as superblocking [4], increase code size dramatically more than [1], and so a comparison against them would make our code size reduction look unrealistically good. Further we will find later in Figure 10.1 that [1] applied within basic blocks yields nearly the same run-time as our provably-optimal-in-run-time within-block scheduler, providing strong verification that [1] is a near-optimal-in-run-time sched-

Name	Description	Type	# lines of code	# Basic Blocks*	# Useful Operations*
adpcm	Converts 16 bit PCM and 4bit ADPCM	tele-com	943	74	467
fft	Discrete Fast Fourier Transforms	tele-com	331	123	750
basicmath	math functions, such as sqrt and sine	auto-moble	315	117	1038
bitcount	counts the number of bits in data	auto-moble	644	87	660
dijkstra	Dijkstra's algorithm for shortest paths	net-work	177	66	411
jpeg	Compress & decom-press jpeg images	con-sumer	51025	11767	94979
blowfish	The encry-ption algorithm	sec-urity	3136	131	1992
rjindael	The AES encryp-tion algorithm	sec-urity	1138	187	3913
sha	The Secure Hash Algorithm	sec-urity	242	82	901
stringsrch	Searches for strings in text	office	3037	187	1448

* *found by examining the output of* PRO64

Table 10.1: Benchmarks. All are in C and from the MiBench suite.

uler. The only compiler technique for explicitly reducing code size that we are aware of for our targeted subset of VLIWs is [3], and so we compare against an implementation of [1] augmented with [3]. See Chapter 2 for a fuller discussion of the related work.

In the rest of this section, Bala is shorthand for our comparison algorithm of Bala et. al. [1] augmented with Freudenberger et. al. [3].

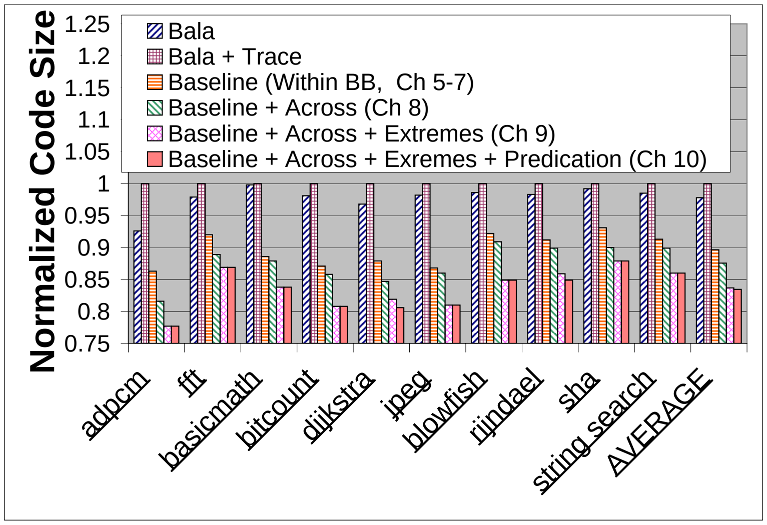
Experiments Figure 10.1 shows the code-size, run-time and compile-time for six methods: (i) Bala applied within basic blocks, (ii) Bala with traces (normalized to

1.0), (iii) our within-trace scheduler applied separately to individual basic blocks, (iv) our within-trace scheduler with across-block analysis, (v) our complete scheduler including the changed optimization criteria at the extremes, and finally (vi) our scheduler with the predication-based methods of Chapter 9. Thus, the second bar represents current methods (the normalized value), the fifth bar represents our general method, and the sixth bar represents an extension of our methods for predicated machines – this sixth bar will not be considered further, because it is not general. By comparing the second and fifth bars, we see that our method reduces the code size by 16.3% compared to the best previous method, while staying within 0.82% of its run-time, a factor of 2.4 of the original compile time [1] In each figure (a, b, and c), the average for all benchmarks is shown as the right-most set of bars for convenience.

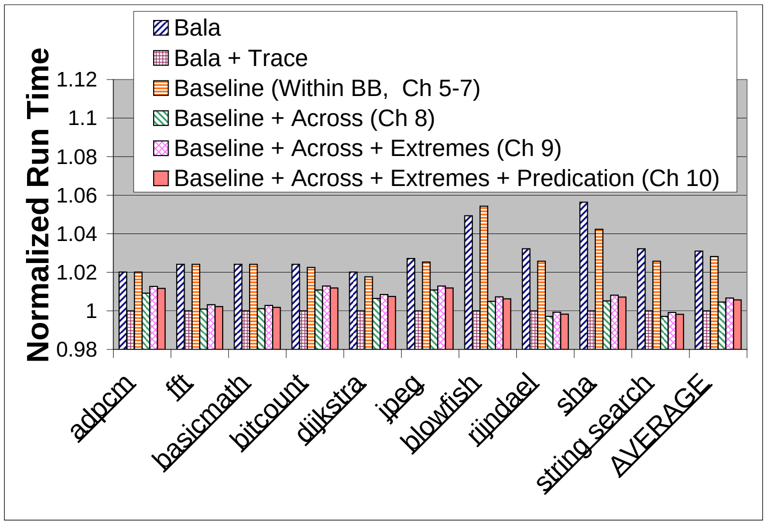
Figure 10.1 also reveals the incremental contributions of our different technologies to the total code size improvement. First, by comparing the first bar versus the third one, we see that our within-block technologies improve code size by 8.1% while maintaining the run-time of current within-block schedulers. In fact the run-time is slightly faster, because the Bala algorithm is greedy. But the closeness of the run-times is strong evidence that the Bala algorithm is doing a good job of finding a minimum run-time solution (albeit with a worse code size). Second, by comparing the third bar versus the fourth, we see that our across-block technologies contribute

[1] Compile-time measurements are inherently approximate, because of variations in coding efficiency. In the case of our implementation, the use of files for communication tends to slow down compilation for all of the tested schemes.

(a)



(b)



(c)

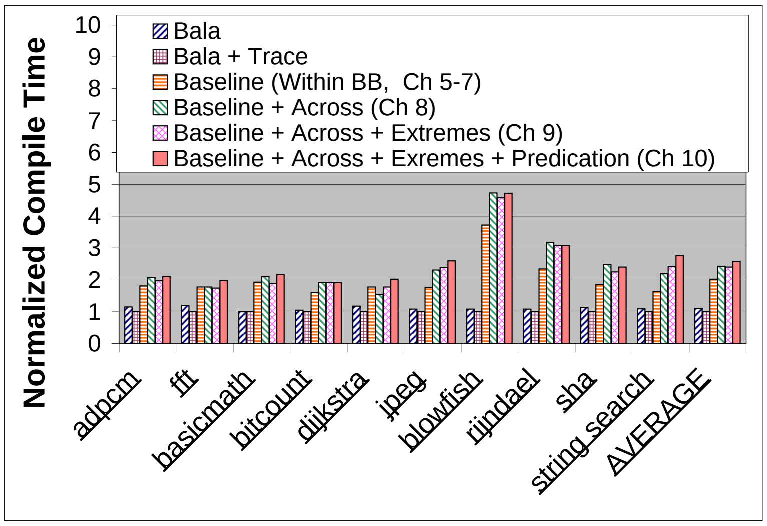


Figure 10.1: (a) Improvement in code size, (b) run-time achieved, and (c) compile-time costs for the methods of Chapters 4-6, 7 and 8, normalized to Bala with traces = 1.0.

an extra 2.1% code size improvement versus our within-block methods. Third, by comparing the fourth bar versus the fifth, we see that changing the optimization criteria at the extremes improves code size by a further 3.9%.

In Figure 10.2, we see the compile time for each benchmark, graphed against the theoretical worst-case compile time. The worst case compile-time is found by assuming that the branch-and-bound scheduler never finishes before its time-out value. The derivation of the time-out value will be described in Figure 10.5. In reality, the time-out is rarely used; but this upper-bound value provides compile-time guarantees, ensuring that our methods never require exponential run-time. From Figure 10.2, we find that most programs are *guaranteed* to finish within a factor of 6 times their original compile time, but in reality finish with just a factor of 2.4 increase. Such a compile-time increase is generally acceptable in embedded systems, because compilation occurs in the factory.

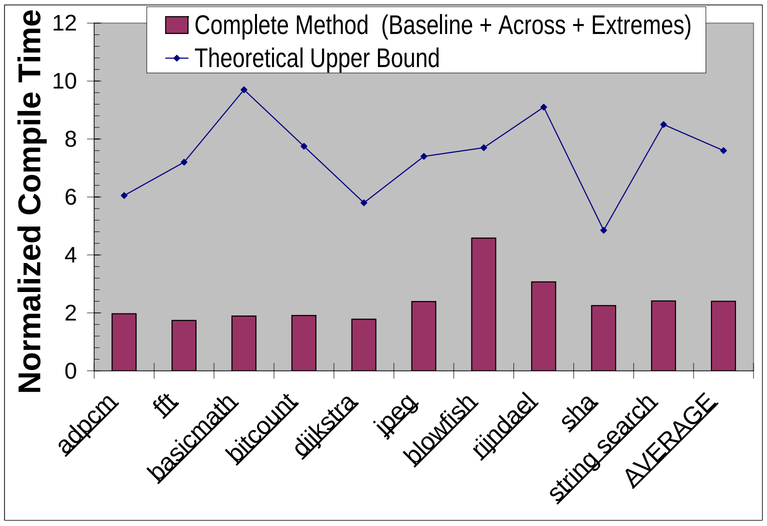
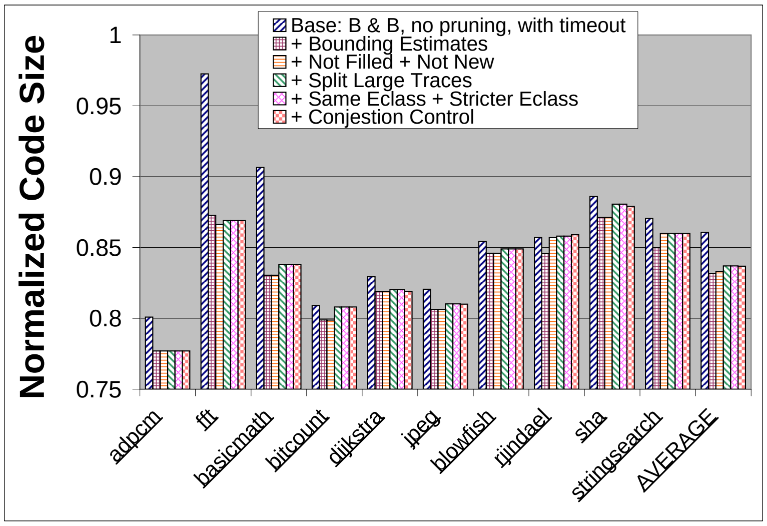


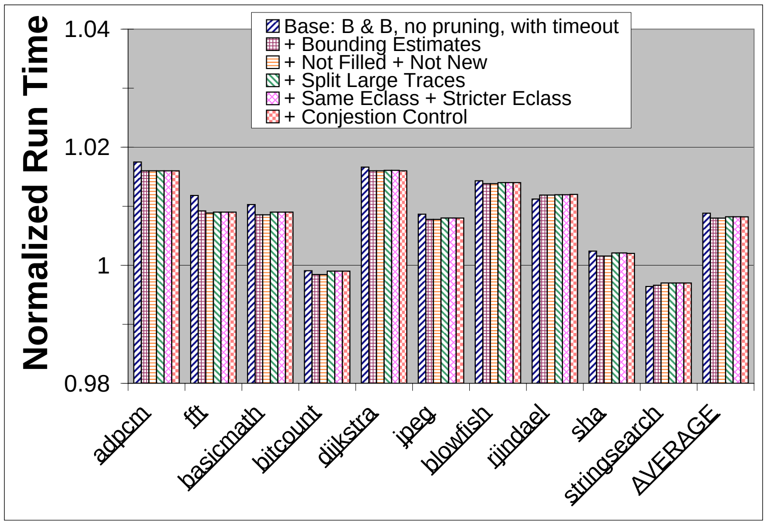
Figure 10.2: Compile-time costs of our final method, compared to the theoretical upper-bound described in Chapter 6, normalized to Bala with traces = 1.0.

Figure 10.3 breaks down the contributions of our various pruning strategies to code size, run-time, and compile time. In Figure 10.3(a), (b) and (c) the right-most bar for each benchmark represents our final algorithm. Figure 10.3(a) indicates a 16.3% savings in code-size, in agreement with the rightmost bar of Figure 10.1(a). Figure 10.3(b) reveals that the pruning techniques have little effect on run-time. This is because minimum-run-time solutions are easier to find than solutions with both minimum run-time and code size. In fact, before performing branch-and-bound, an initial bound for the best-schedule-found-so-far is obtained through a greedy search similar to Bala, and Figure 10.1(b) has already shown that Bala produces a comparable run-time to our approach. Therefore there is little chance for improvement in run-time. Instead, the effect of pruning strategies is to reduce the compile time, and as a consequence, to allow a larger portion of the design space to be searched before timing out – perhaps leading to the discovery of a more-compact schedule. Finally, Figure 10.3(c) reveals that the compile-time overhead of our algorithm benefits substantially from pruning.

Figure 10.3 further reveals that, while the pruning techniques have an effect on code size (Figure 10.3a) and compile time (Figure 10.3c), they dot not affect run-time (Figure 10.3b). This is because minimum-run-time solutions are easier to find than are solutions with both minimum run-time and code size. In fact, before performing branch and bound, an initial bound for the best-schedule-found-so-far is obtained through a greedy search similar to Bala, and Figure 10.1(b) has already shown that Bala produces a comparable run-time to our approach. Therefore there is little chance for improvement in run-time. Instead, the effect of pruning strategies is to reduce the compile time, and as a consequence, to allow a larger portion of the design space to be searched before timing out – perhaps leading to the discovery of a more-compact schedule.

We now examine the measured effect of each of the pruning techniques. The first bar represents our algorithm without any pruning, except for the time-out strategy. The impact of the time-out heuristic cannot be measured, since without it a few large traces did not finish even in days, although most traces finished within the time-out of a few seconds. Only a small number of traces timed out after 1 second, and increasing the time-out did not significantly reduce the number of time-outs or improve the code size. This basic approach demonstrates a 15.3% code size improvement and a factor of 8.6 increase in compile time. The second bar represents the addition of lower bound pruning to the base algorithm. Where as the first bar does not consider the cost of the as-yet-unscheduled instructions, the algorithm of the second bar considers lower bounds on the remaining cost, thus providing a tighter bound for pruning. This pruning technique demonstrates a further 2.6% decrease in code size, and 30.6% reduction in compile time. The third bar indicates the pruning technique whereby certain Instruction Group choices are not considered, because they are either redundant or provably inferior to another possible schedule. This reduces the code size by a further 0.19% and the compile time by 24.4 %. The fourth bar reveals that splitting large traces into more manageable pieces (of 48 instructions each) *increases* the code size by 0.46% over the third bar, but reduces the compile time by 45.8%. Splitting large blocks is a non-optimal heuristic, so it

(a) 

(b) 

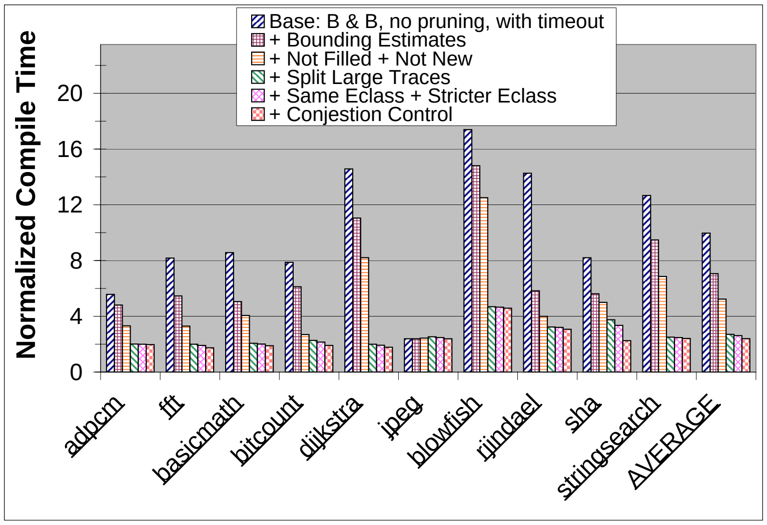
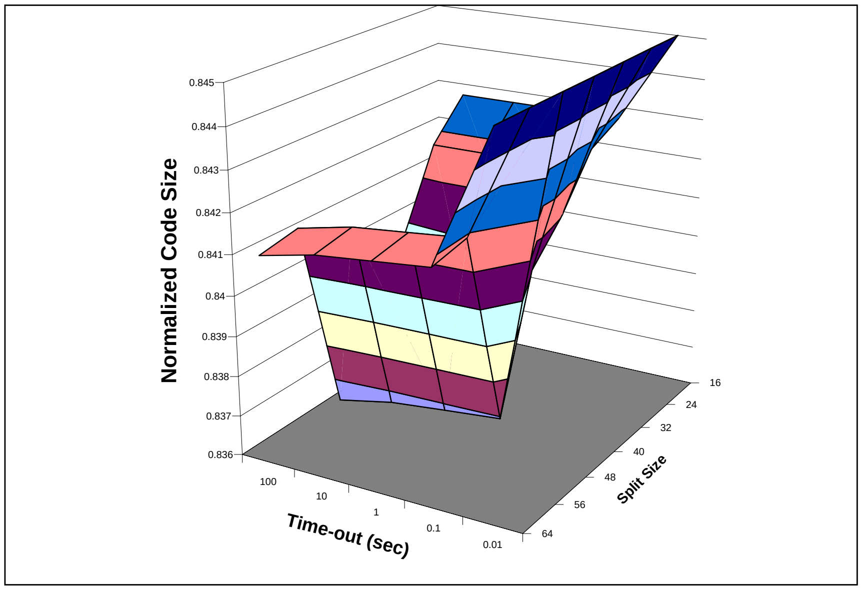
(c) 

Figure 10.3: The effect of our pruning strategies upon (a) code size, (b) run-time, and (c) compile time, normalized to Bala with traces (Bala does not use pruning). Our proposed algorithm is represented by the left-most bar.

is possible to increase the code size. Yet, since this heuristic has such a positive effect on compile time, a designer may choose to compile with this flag. Next, the fifth bar considers the effect of adding in Equivalence class analysis, which removes redundancies among possible instruction groups. This technique had no measured effect on code size, but decreased compile time by 3.6%. Lastly, the sixth bar represents our final algorithm, which uses all of the pruning techniques. This final bar shows the effect of congestion control, where edges are inserted between constrained, FIXED_CYCLE instructions. This pruning technique reduces code size by only 0.02%, but reduces the compile time by an additional 9.18%.

Figure 10.4 examines the interaction between our two primary *non-optimal* pruning methods: 1) timing-out and 2) splitting traces. Figure 10.4 (a) presents a 2-dimensional surface plot of the code size that results from a variety of time-out and split values. For any given configuration of time-out and split-size, the reported code size is the measured average across all of our benchmarks. Figure 10.4 (b) presents a similar plot for compile time. The shape of Figure 10.4 (b) indicates that compile time grows exponentially with both the split size and the time-out value. It also shows that the compile time can be reduced by *either* reducing the split size or the time-out value, so that they must be chose together. Figure 10.4 (a) indicates that the code size is relatively flat for time-out values greater than 10 ms; at the same time, however, the figure does also show that there is a *slight* benefit to using a split size of 48 instructions. There is also an even smaller code-size benefit from choosing a time-out greater than 1 second. Yet a quick glance at Figure 10.4

(a)



(b)

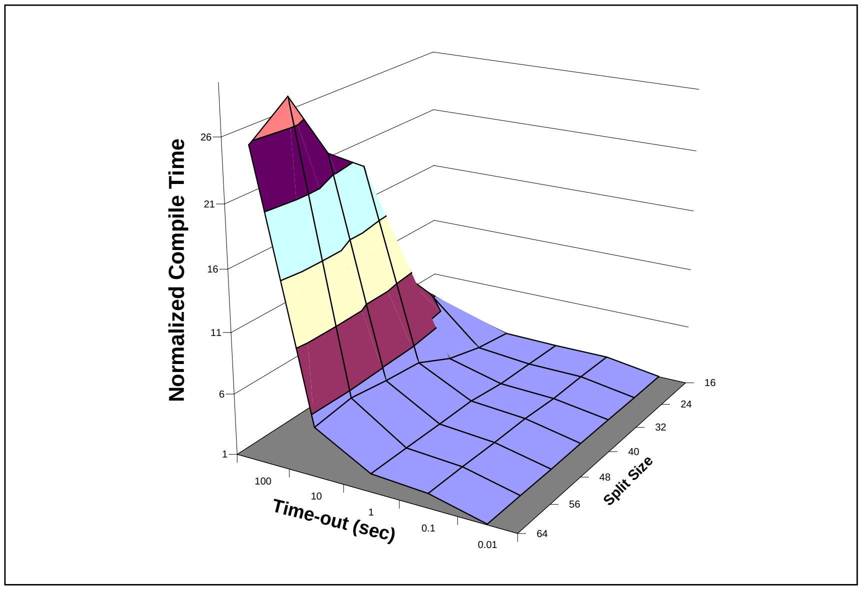


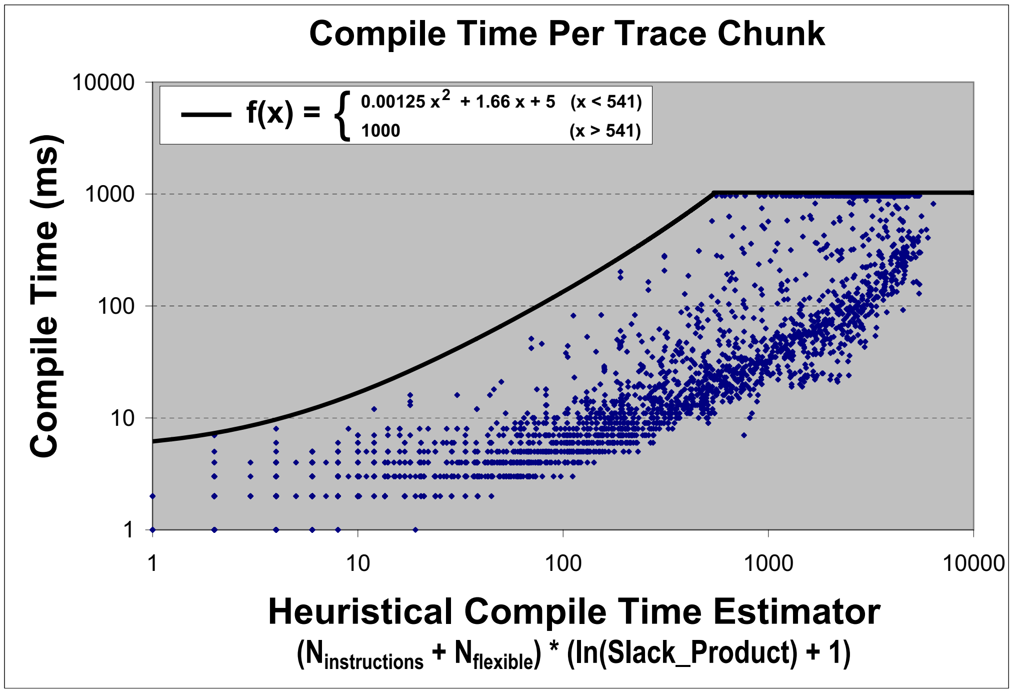
Figure 10.4: Determining the proper values for the time-out and split-size heuristics. The time-out and split-size values are inter-dependent, so must be simultaneously evaluated. In (a), the code-size improvement of our full method, *averaged* across all of our benchmarks, is plotted versus the time-out and split-size values used. In (b), the average compile-time of our method is plotted. (The run-time is not plotted because it is flat.)

(b) demonstrates that the compile time grows rapidly for time-out greater than 1 second. Therefore, we have chosen a time-out value of 1 second and a split size of 48 instructions. As presented in Chapter 6, however, the 1-second time-out value is an upper limit; most traces are given a smaller time-out value, depending on the length and other characteristics of the trace.

We now describe the derivation of the trace-dependent time-out function. To begin, Figure 10.5(a) displays the actual compile time of every trace chunk from every benchmark. Each of the 8,390 data points represents one trace chunk: the y-value being the chunk's measured compile time and the x-value being computed as a simple function of the chunk's properties. We have developed this function so as to estimate a chunk's compile time before scheduling it. This function is the product of two terms. The first term is the number of instructions, $N_{instructions}$, plus the number of instructions not on a fixed cycle, $N_{flexible}$. The intuition here is that instructions with scheduling flexibility will increase the search space, and so they affect the compile time more than fixed instructions. The second term also attempts to measure the size of the search space: it is the natural log of the product of all slacks in the DFG, plus one. In Figure 10.5(b), this same estimator function is used for the x-value of the data points, but the y-value is the time at which the final solution was found. Therefore, the data points in Figure 10.5(b) are lower than in (a), because a branch-and-bound algorithm *may* search for a long time without finding a better solution than one visited early in the search.

Figure 10.5 also illustrates our time-out function. By fitting a parabola through

(a)



(b)

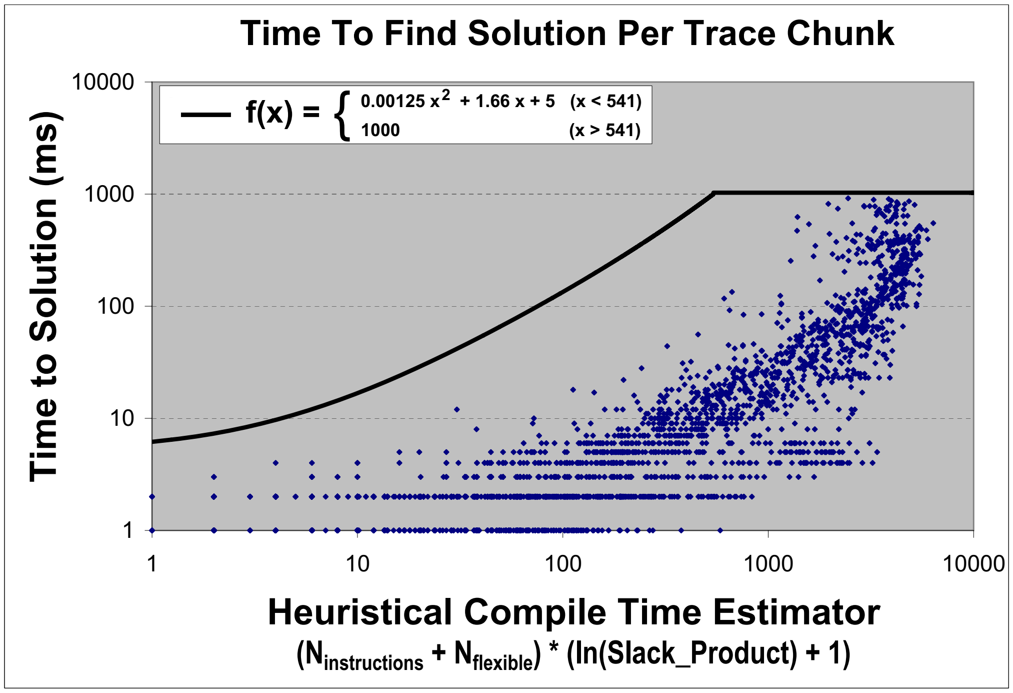


Figure 10.5: Determining the proper time-out function. In (a), the actual measured compile time for each trace chunk is plotted versus a simple heuristic estimator of compile time. In (b) the time when the final solution was reached is given. In both figures, our proposed function is also plotted, chosen by fitting a parabola two the upper 2% of data points, while requiring the parabola's y-intercept to be 5 ms.

the top 2% of data points in Figure 10.5(a), while requiring the y-intercept to be 5 ms, we arrive at the time-out function, f(x), plotted in Figure 10.5. The y-intercept restriction is needed so as to ensure that the smallest chunks are given sufficient time to compile, which turns out to be 5 ms. In this figure, we also see that the time-out function is clipped at 1000 ms. Examining Figure 10.5(a), we notice that almost all trace chunks finish compilation without timing out (95%). In Figure 10.5(b), we further observe that an even larger number of chunks will have found their final solution before timing out. As a result, the time-out function has a marginal impact on the quality of our results, but a major impact on the quality of our compile-time guarantees.

Figures 10.6 and Figure 10.7 illustrate how the thresholds of Chapter 8 are chosen. Some explanation of the meaning of these thresholds is warranted. If Threshold_Code_Size = X this indicates that only code size matters for the set of least frequent traces, whose combined execution time is X% of the total execution time of the program. Similarly, if Threshold_Run_Time = Y this indicates that only run-time matters for the set of most frequent traces, whose combined execution time is (100 - Y)% of the total execution time of the program. The sense of this definition is reversed so that the two thresholds can be understood together, with common units, as in Figure 8.1. Thus for both thresholds, a higher value corresponds to a higher preference of code size versus run-time. When no thresholds are used (as in Chapter 7, and corresponding to the fourth bar of Figure 10.1), the effective values of these thresholds could be considered to be: Threshold_Code_Size = 0% and Threshold_Run_Time = 100%.

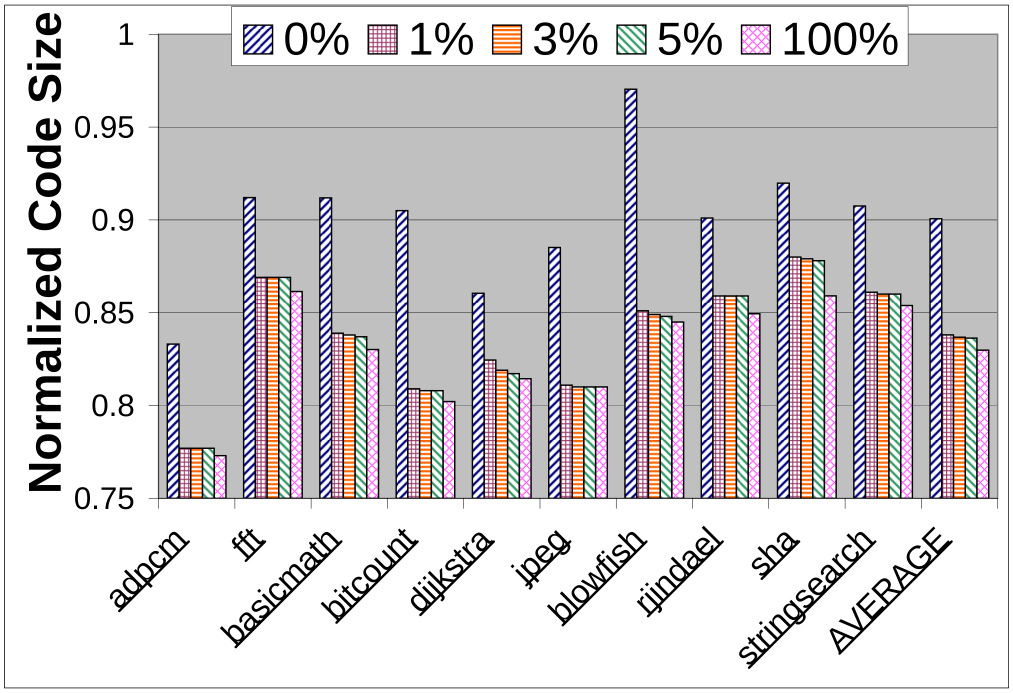
Figure 10.6 illustrates our method for determining the optimal value for the Threshold_Code_Size parameter. In this figure, the Threshold_Run_Time parameter is set to its optimal value (15%). We see from this figure that choosing 3% for Threshold_Code_Size achieves most of the code size improvement of limiting compensation code, while at the same time achieving most of the run-time benefit of trace scheduling. This is because most of the execution time is spent in a small number of traces.

Similarly, Figure 10.7 depicts the effect of the Threshold_Run_Time parameter, with Threshold_Code_Size set its optimal value (3%). This figure shows that choosing a value of 15% for Threshold_Run_Time maintains most of the code size improvement of the code-size-only schedule, without a serious impact on run-time.

It is interesting to see how our results change with VLIW width. Since IA-64 packs 3 instructions into a bundle, it is possible to also model a 3-wide VLIW, instead of the 6-wide machine above, and run the code on Itanium hardware. Figure 10.8 describes the results of our method on a 3-wide VLIW. By comparing Figures 10.8 and 10.1, we make the following two observations. First, the average code size reduction for a 3-wide VLIW is 13.6%, which is 86% of the improvement found in the 6-wide VLIW. Hence, the code size improvement of our method scales reasonably well. This information also suggests that our technique may have a somewhat larger effect on even wider VLIWs. Second, the basic shapes of the two figures are similar.

It is also interesting to see the relative proportion of trace lengths in our benchmarks. Figure 10.9 divides the traces according to their number of useful

(a)



(b)

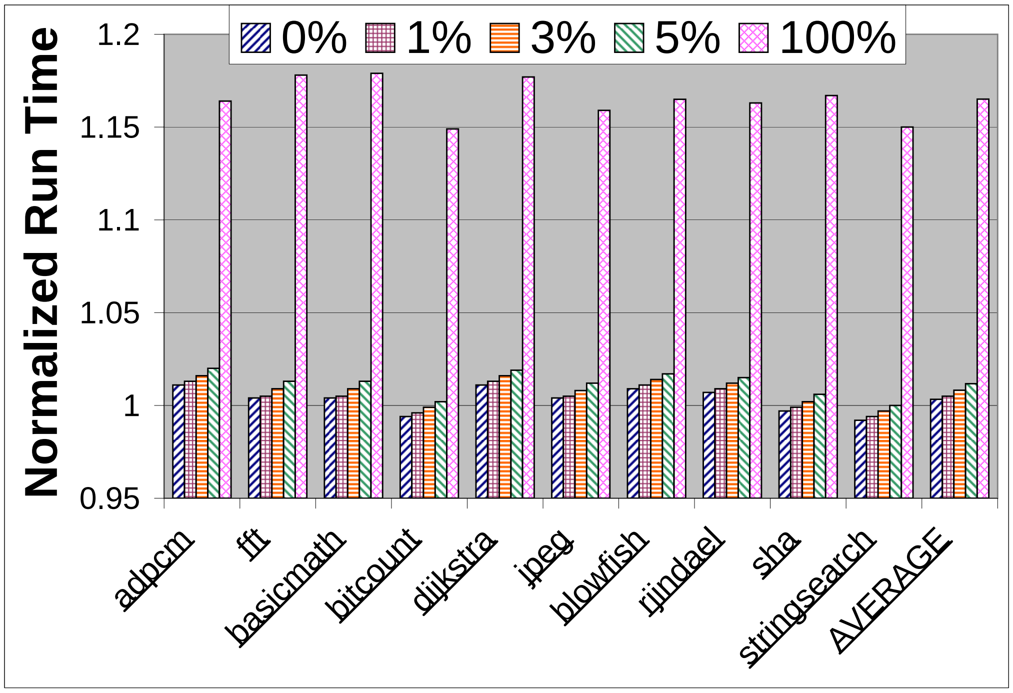
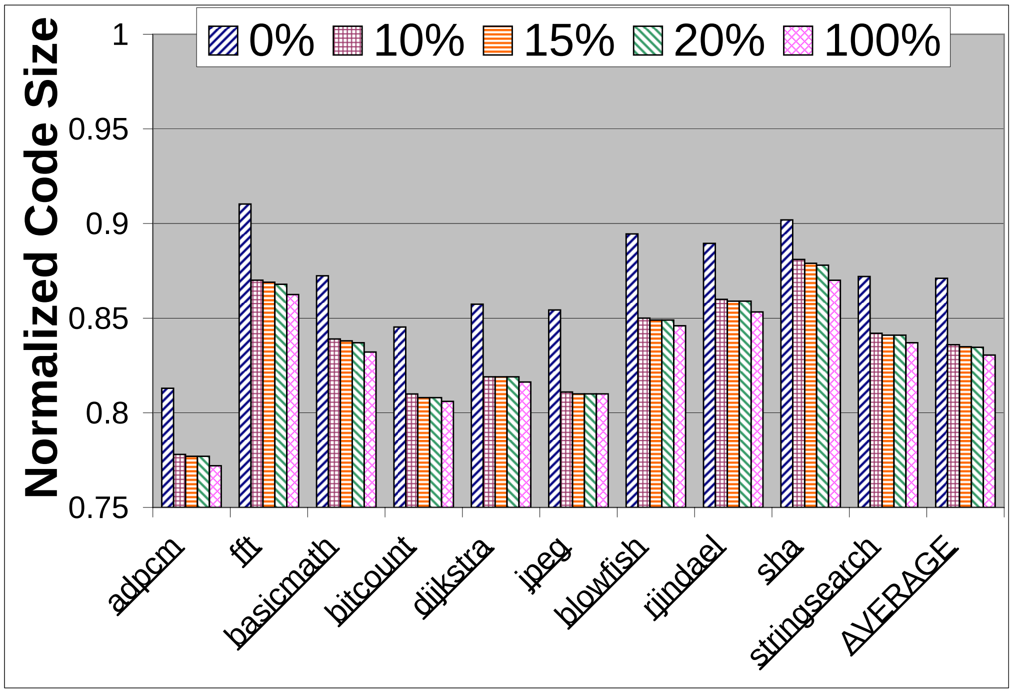


Figure 10.6: The results of varying the Threshold_Code_Size parameter, upon (a) code size and (b) run time, normalized to Bala with traces. (Threshold_Run_Time is held constant).

(a)



(b)

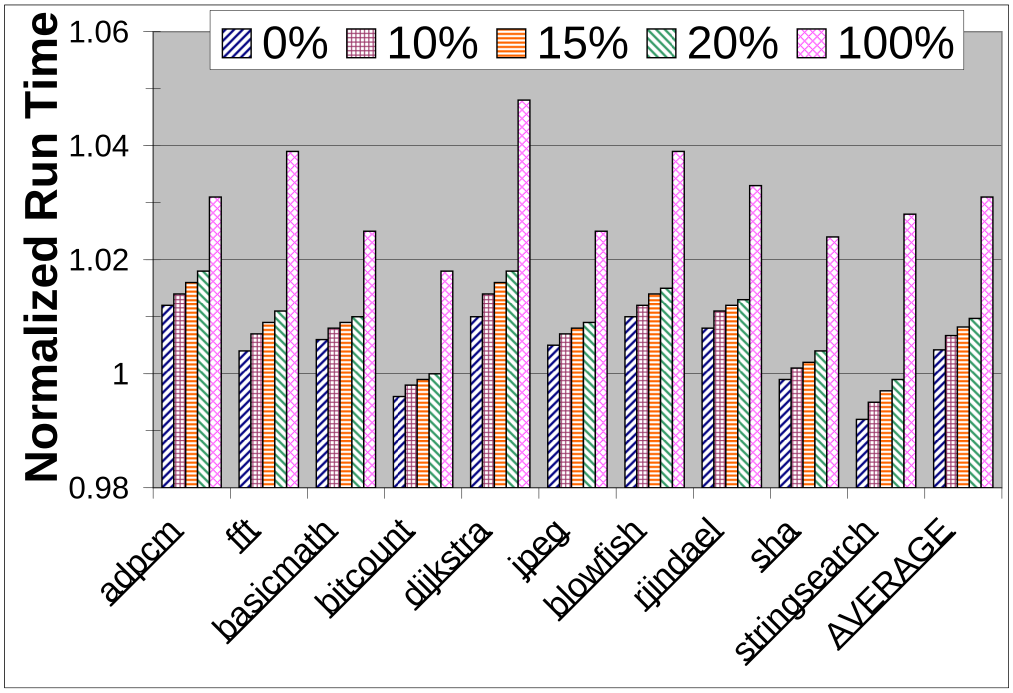
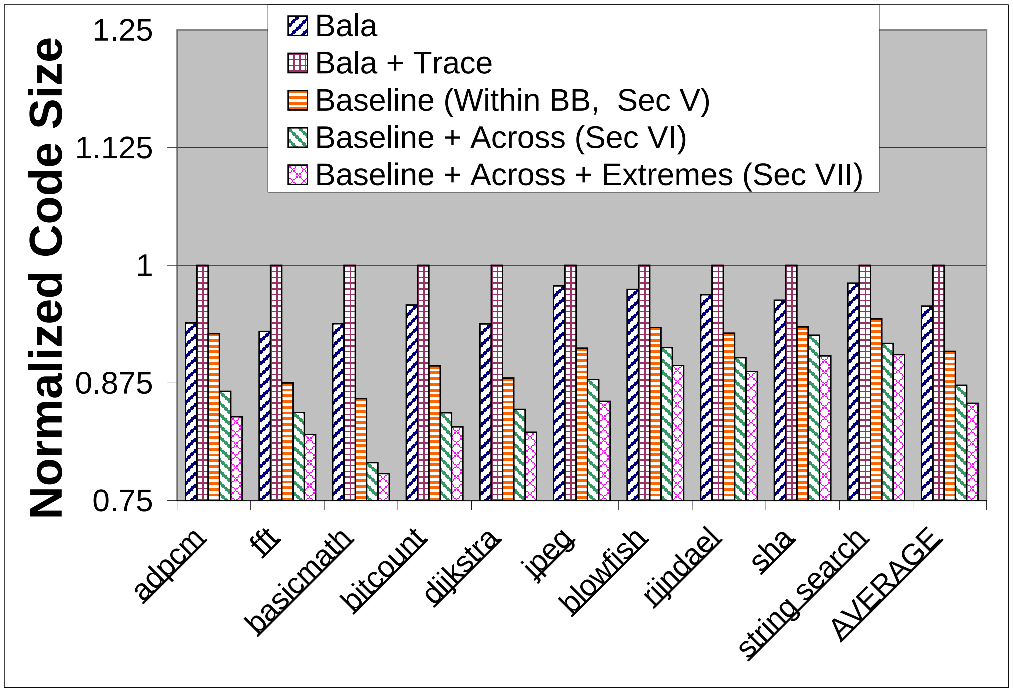


Figure 10.7: The results of varying the Threshold_Run_Time parameter, upon (a) code size and (b) run time, normalized to Bala with traces. (Threshold_Code_Size is held constant).

(a)



(b)

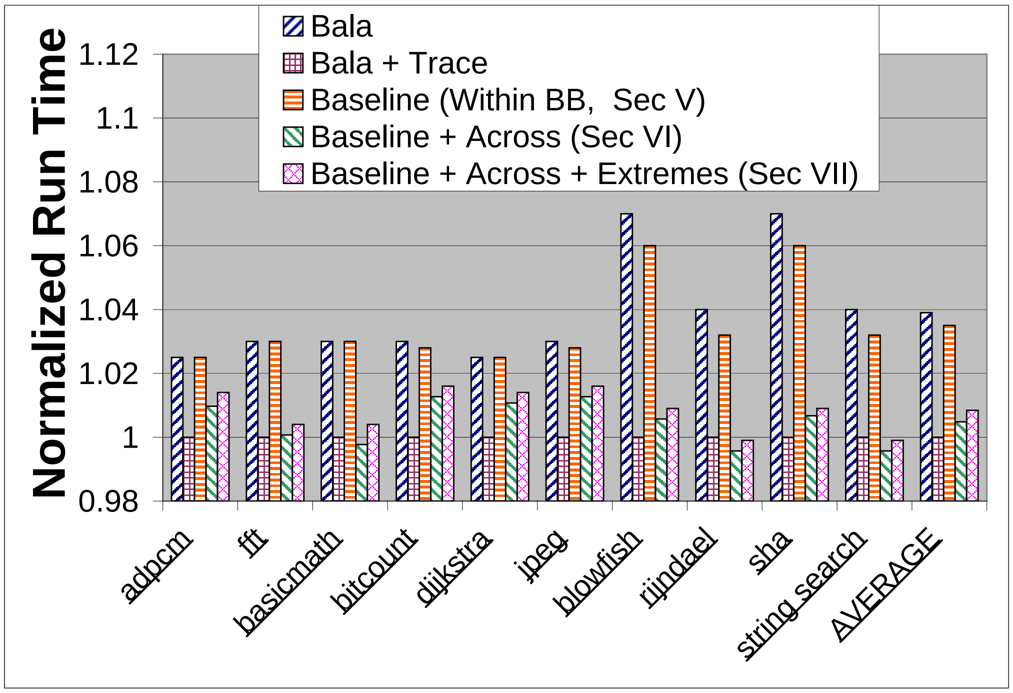


Figure 10.8: The results for our complete technique on a 3-wide VLIW for (a) code size and (b) run-time, normalized to Bala for a 3-wide VLIW with traces

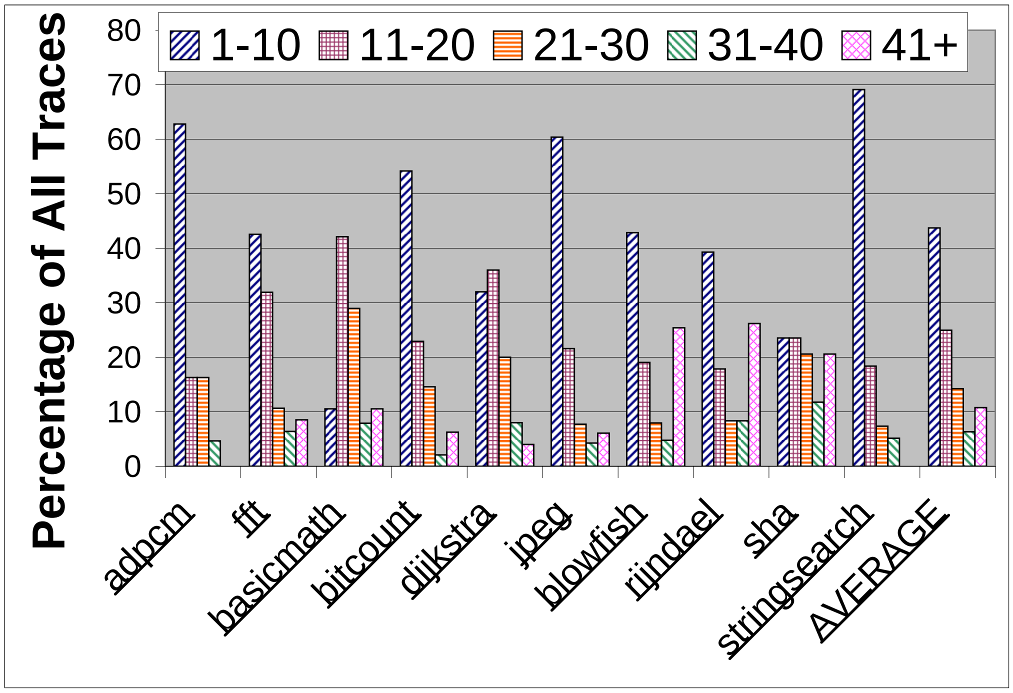


Figure 10.9: The distribution of trace lengths (not counting NOPS) for each benchmark.

instructions, and gives the percentage of traces in each range. We see that 69% of traces are less than 21 instructions long. These traces are likely to schedule quickly without the need of non-optimal heuristics.

10.1 Cache Behavior

The reduced code size of our approach will lead to an improved I-cache behavior. On our test platform (an Itanium Merced), this effect is very slight, because the Itanium is a server with a large I-cache. For embedded systems, however, the I-cache is typically smaller and a modest run-time benefit of around 1% is achievable.

Additionally, this I-cache improvement extends the scope of machines to which our approach applies. Since many compressed VLIWs will expand the long-words before inserting them into the I-cache, these machines will also experience the run-time benefits of our method's I-cache improvement.

The process of collecting cache numbers is now described. First, a trace is created of all instruction addresses fetched during the execution of each benchmark. Next, the Dinero IV cache simulator [38] is used to identify the number of I-cache misses with a 16 kilobyte, 4-way set associative instruction cache, with 32 byte blocks. This configuration corresponds to the Itanium that we use for finding our execution times. The Itanium also has a 96 kilobyte, 6-way set associative, unified, Level 2 cache, with an access time of 6 cycles. [39] Instead of also measuring the data cache, we make the simplifying assumption that instruction fetches never miss in the Level 2 cache. This assumption is reasonable, since I-caches tend to have better hit rates than data caches. At a clock frequency of 750 MHz, and a miss penalty of 6 cycles, each Level 1 I-cache miss will cost 8ns. Therefore, the amount of time that our Itanium spends waiting on the I-cache, $time_{Itanium}$, is roughly 8ns $\times$ the number of misses, as found from Dinero. This number is usually a small part of the program's execution time. By subtracting this number from the observed run-time, we have the theoretical run-time of our method on an Itanium with a perfect I-cache hit rate.

To generate the I-cache figures shown in the results, Dinero was rerun with different caches more typical of embedded systems. In Embedded systems, there may often not be an L2 cache, and so the miss penalties from the L1 cache are perhaps 25 cycles. When scaled to a 750 MHz machine, the miss penalty is 33ns. Therefore, we can estimate the run-time on different caches by computing: $Run -$

$time_{parameters} = Run - time_{Itanium} - (8ns \times \#Misses_{Itanium\ parameters}) + (33ns \times \#Misses_{test\ parameters})$

Chapter 11

Conclusions

Code size is an important concern in many embedded systems. A method is described for instruction scheduling to reduce code size for a particular subset of VLIWs without sacrificing run-time. The within-trace scheduler is based on optimal approaches, but uses branch and bound methods, as well as heuristics to reduce the compile time. Further, we present an across-block analyzer that runs before within-trace scheduling; it identifies which across-block instruction moves are likely to increase code size, and disallows them from movement subsequently. Finally our method is configurable to target either only code size, or only run-time, instead of both, which is useful in the least frequent and most frequent traces, respectively.

The intellectual novelty of our scheme is seen in the following four contributions. First, unlike existing schedulers, before doing within-trace scheduling, our method places constraints on the movement of certain instructions across basic blocks if the move will likely increase code size. Second, our method is unique in that it does a preliminary scheduling of basic blocks to estimate which instruction moves are likely to increase code size, before doing within-trace scheduling. Third, we are the first to develop a back-tracking technique targeted for code size, and to develop a series of innovative pruning techniques unique to a search for a minimum code size solution. Fourth, our method is unique in targeting different objectives for traces of different frequencies – code size only for infrequent traces; run-time only for frequent traces; and both code size and run-time for traces of intermediate frequency.

Our approach improves the code size by an average of 16% over existing methods with trace scheduling, while still extracting nearly the same speedup that trace scheduling achieves. Without trace scheduling, our approach reduces the code size by 8.2% with a small *improvement* in execution time.

Appendix A

Derivation of the Finite State Automatons for our 6-wide and 3-wide VLIWs

This Chapter describes the details involved in creating Finite State Automatons (FSAs) for our test VLIWs. A presentation of how these FSAs are then used by the compiler is found in Chapter 4. An FSA describes a syntax of legal sentences for a given grammar. Following the approach of [1], we define one grammar that corresponds to all valid instruction schedules in our 6-wide VLIW, and another grammar for legal instruction schedules in our 3-wide VLIW. Although directly based on the methods of [1], the process of deriving and simplifying an FSA for a specific processor is still rather involved, and so must be presented for completeness. Though highly machine-specific, this chapter also contains a number of insights that are more generally applicable, especially for template-based CPUs.

Limitations on instruction scheduling are the result of resource constraints, such as the VLIW width, the physical number of functional units of a given type within the CPU, or template restrictions. Since these limitations on instruction scheduling determine the rules of our grammar, any discussion of FSA formulation must be prefaced with a treatment of the resource constraints of the system.

A.1 Resource Constraints of our 6-wide VLIW

This section explores the resource constraints of our 6-wide VLIW. Since our implementation is conducted using an Itanium[] compiler, We simulate a 6-wide VLIW with the hardware of an Itanium. Unlike the Itanium, however, we do not allow variable long-word-widths, since the EPIC[] architecture of the Itanium is not typical of many embedded systems. Yet similar FSAs can be generated for EPICs as well.

The execution units present in our test machine are those of the Itanium. Chapter 4, Section 4.2 presents the nomenclature of these functional units, and should be read before this Appendix. In particular, we note the distinction between *general* and *restricted* functional units , and the *general* and *specialized* instruction types; the meaning of a general execution unit and a general instruction are reversed from each other: the restricted i-unit can only execute general I-type instructions, while specialized i-type instructions require the general I-type execution unit. Therefore an i-type instruction may only execute the general I-unit, where as a general I-type instruction may use either the i-unit or the general I-units. Section 4.2 also introduces a nomenclature that allow for lower-case letters to indicate restricted functional units and/or specialized instructions.

Templates With this nomenclature, we may proceed to a consideration of the templates available in our 6-wide VLIW. Such a discussion must begin by considering the templates provided by the IA-64, shown in Figure A.1. Because IA-64 instructions are *bundled* into groups of three, each template in this figure contains

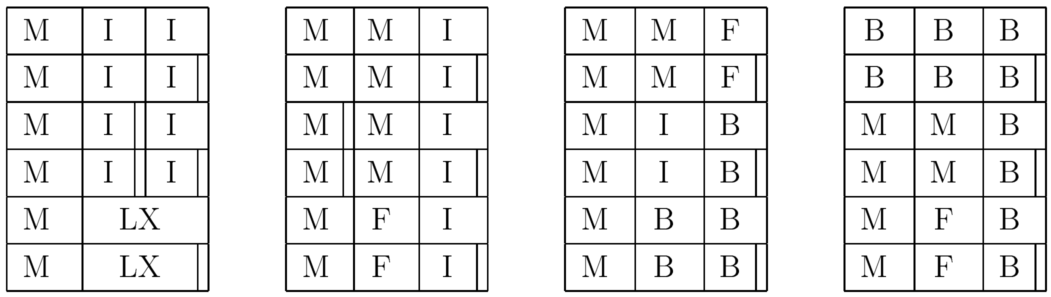


Figure A.1: The 24 templates of the IA64. Thin vertical bars represent stop bits.

three columns (called *slots*). For a bundle to be compatible with the IA-64 ISA, it must match one of the 24 predefined templates in this figure. A particular bundle matches a particular template if all three of the instructions within that bundle may be assigned to the execution unit indicated by the corresponding slot in the template. For instance, the first template in this figure is “MII”, indicating that the first slot will be issued to an M-unit, while the second and third slots will issue to both of the I-units. A bundle comprised of an M-type instruction followed by two I-type instructions would match to this “MII” template. We note, however, that such a bundle is not the only combination of instructions types that matches the “MII” template; for example, a bundle of all A-type instructions also matches this template, since A-type instructions may execute on either M-units or I-units.

Two additional detail of Figure A.1 are now considered. First, there are two templates that allow for LX-type instructions. Because LX-type instructions are wider and utilize both an I-unit and an F-unit, the LX-type instruction occupies two slots in these templates. Second,Many of the templates in this figure contain thin vertical bars. These bars indicate *stop bits.* The Itanium is an EPIC architecture; unlike fixed width VLIWs, EPICs provide for explicit parallelism, through the mechanism of stop bits. Stop bits are used to divide code into parallel regions, or *instruction groups*. Some of the templates in Figure A.1 contain two stop bits. This indicates that not all instructions contained within a single bundle need not be parallel even with each other. Also, some templates in the figure contain no stop bits. In theory, these templates allow for instruction groups of unlimited size. In reality, the Itanium has a fetch window of two bundles, so that longer instruction groups will be executed on successive cycles. We also note that the 24 templates of Figure A.1 can be viewed as 12 *template pairs*, each pair differing only in the stop bit after the third slot. Although more could be said regarding stop bits, is unnecessary for the present purpose, because our fixed-width VLIW makes little use of stop bits.

Having described the IA-64 templates, we now derive the *6-wide templates* for our test VLIW. Two observations will allow us to reduce the number of possible templates from 24 to 18. First, we observe that instructions within a bundle will always be executable in parallel in our fixed-width VLIW. Therefore, our compiler can never legally schedule an instruction bundle that matches one of the templates with an internal stop bit. Second, we observe that a 6-wide VLIW will require exactly 2 bundles, but special *Itanium* restrictions prevent the "MMF" template pair from ever executing in parallel with a second bundle. To begin with, the resource restrictions already described remove most possible combinations for the "MMF" template pair. Since these templates already uses both M-units, they could only execute in parallel with templates that uses no M-units. But the only such templates are the "BBB" template pair. Therefore, the only possible 6-wide combinations are

"BBBMMF" and "MMFBBB". Itanium imposes additional constraints that remove even these possibilities, however. As stated in [], Itanium always stalls after either a "BBB" or "MMF" template, regardless of whether these templates end with a stop bit. Therefore the Itanium will stall after the first bundle of either of the "BBBMMF" or "MMFBBB" two-bundle combinations. Since our test processor is chosen to be a fixed-width VLIW that matches the Itanium in all other aspects (and since our experimentation is performed on an Itanium), we must obey these additional Itanium restrictions. Therefore the "MMF" template pair may also be removed from consideration. Figure A.2 presents the 18 templates that remain after the above considerations.

Not every bundle can choose any of these 18 templates, however, because our parallel regions are always two bundles wide (*i.e.,* . six instructions). This means that a stop bit will be needed after every second bundle. Therefore, the first bundle of each 6-wide long-word must use a template that does not contain a stop bit, while the second bundle must match to one of the templates that does end in a stop bit. It is therefore natural to rearrange the 18 templates according to whether they end in a stop bit, as shown in Figure A.3. When rearranged in this fashion, the left column represents the template choices for the first bundle and the right column lists the choices for the second bundle.

Figure A.3 may be further reduced by considering Itanium resource requirements. For the first bundle, the "MBB" and "BBB" templates may be removed, because Itanium always stalls after either of these templates. Once these are removed, all remaining first-bundle templates begin with an "M." As a result, only

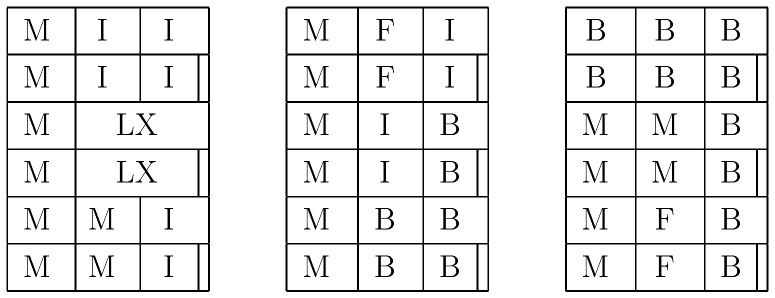


Figure A.2: The 18 templates available to a 6-wide VLIW. Templates with internal stop bits, and the "MMF" template are disallowed.

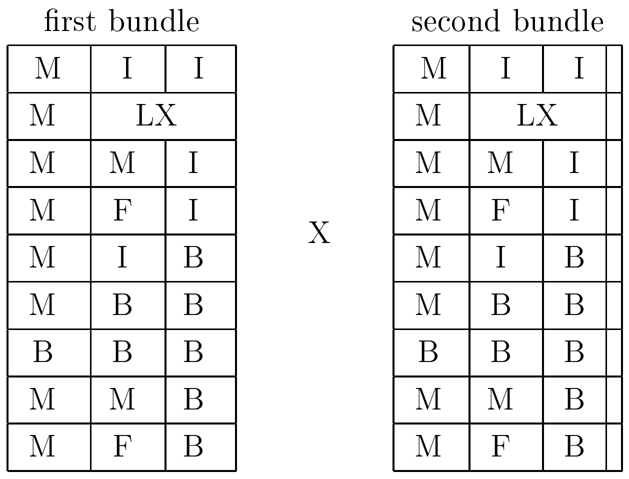


Figure A.3: The 18 templates, rearranged according to bundle. Since a 6-wide VLIW will have a single stop bit at the end of each long-word, 9 of the templates are available for the first bundle, and the other 9 are available for the second bundle.

one M-unit (at most) remains for the second bundle. Therefore, the "MMI" and "MMB" templates, which need two M-units, cannot be used.

Figure A.4 presents the final template choices for the bundles of our 3-wide machine. This figure would appear to indicate that there are 49 possible combinations, but there are in fact far fewer. For instance, the "MIIMII" combination is impossible because it would require four I-units. Enumerating the 49 possibilities, we find that 22 exceed the resource constraints of the Itanium, leaving the 27 templates presented in Figure A.5. In this figure, and in the paragraphs to follow, a first bundle

M	I	I
M	LX	
M	M	I
M	F	I
M	I	B
M	M	B
M	F	B

X

second bundle

M	I	I	
M	LX		
M	F	I	
M	I	B	
M	B	B	
B	B	B	
M	F	B	

Figure A.4: The 14 templates that remain after considering Itanium template restrictions. Because Itanium inserts stop bits after the “BBB” and “MBB” templates, these templates cannot use the first bundle. As a result, the 7 remaining templates for the first bundle all use at least one “M” unit. Therefore, since only two “M” Units exist, the second slot can only use, at most, one “M” Unit. As a result the “MMI” and “MMB” templates cannot be used for the second bundle.

template now refers to a 6-wide template used in our hypothetical machine, rather than to the IA-64’s 3-wide templates. Both usages of the word are correct, but must be distinguished from each other: the underlying Itanium hardware uses 3-wide templates, but the compiler of our hypothetical machine *need not be concerned with the underlying hardware* – it schedules 6-wide instruction groups directly to the 6-wide templates of Figure A.5. The reason that that we do not need to be concerned with 3-wide templates is because the above analysis has already pre-computed this information to determine the 27 possible templates. Figure A.5 is the final result that is reproduced in Chapter 4 as Figure 4.2.

Although Figure A.5 is the final result included in Chapter 4, yet at the lower-implementation level, one additional step is useful. Finally, Figure A.6 summarizes the execution units used in each of the 27 templates. This resource-assignment map will be important in determining the FSA in Section **??**. Figure A.6 illustrate

M	I	I	M	B	B
M	I	I	B	B	B
M	I	I	M	F	B
M	LX		M	LX	
M	LX		M	F	I
M	LX		M	I	B
M	LX		M	B	B
M	LX		B	B	B
M	LX		M	F	B

M	M	I	B	B	B
M	F	I	M	LX	
M	F	I	M	F	I
M	F	I	M	I	B
M	F	I	M	B	B
M	F	I	B	B	B
M	F	I	M	F	B
M	I	B	M	LX	
M	I	B	M	F	I

M	I	B	M	I	B
M	I	B	M	B	B
M	I	B	M	F	B
M	F	B	M	I	I
M	F	B	M	LX	
M	F	B	M	F	I
M	F	B	M	I	B
M	F	B	M	B	B
M	F	B	M	F	B

Figure A.5: The 27 templates available for our 6-wide machine. Combining the bundles from Figure A.4 produces 49 possible combinations (7 x 7). Yet 22 of these can be dropped because they exceed the resource constraints (*e.g.*, "MIIMII" uses too many "I" Units).

M	I	i	m	B_1	$B_{0/2}$[a]
M	I	i	B_0	B_1	B_2
M	I	i	m	f[b]	$B_{0/2}$[a]
M	F	I	m	f	i
M	F	I	m	f	i
M	F	I	m	i	$B_{0/2}$[a]
M	F	I	m	B_1	$B_{0/2}$[a]
M	F	I	B_0	B_1	B_2
M	F	I	m	f	$B_{0/2}$[a]

M	m	I	B_0	B_1	B_2
M	F	I	m	f	i
M	F	I	m	f	i
M	F	I	m	i	$B_{0/2}$[a]
M	F	I	m	B_1	$B_{0/2}$[a]
M	F	I	B_0	B_1	B_2
M	F	I	m	f	B
M	I	B_0[d]	m	f	i
M	I	B_0[d]	m	f[b]	i

M	I	B_0[d]	m	i	B_2
M	I	B_0[d]	m	B_1	B_2
M	I	B_0[d]	m	f[b]	B_2
M	F	B_0[d]	m	I	i
M	F	B_0[d]	m	f	i
M	F	B_0[d]	m	f	i[c]
M	F	B_0[d]	m	I	B_2
M	F	B_0[d]	m	B_1	B_2
M	F	B_0[d]	m	f	B_2

[a] *Itanium assigns this slot to B_0 if instruction is a brp or a nop. Otherwise, it is assigned to B_2.*
[b] *Itanium assigns this slot to the restricted F Unit, even though the unrestricted unit is available.*
[c] *Itanium assigns this slot to the restricted I Unit, even though the unrestricted unit is available.*
[d] *Itanium only allows a brp instruction or a nop instruction to fill this slot.*

Figure A.6: The slot assignment of the 27 templates to the Itanium resources. In this figure, the lower-case letters (m, i, and f) are used to indicate the restricted functional units that the Itanium literature refers to as (M_1, I_1, and F_1). In general, the leftmost instance of an instruction type within a given template will be assigned to the general functional unit of that type. Exceptions to this rule are noted in the figure. Also note that the LX instruction consumes both an "F" and an "I" resource.

four characteristics of the Itanium. First, LX-type instructions are shown to use both an F-unit and and I-unit. Second, the assignment of general execution units is usually left to right. For instance, in the first template ("MIIMBB"), the resource assignment is "MIimBB", indicating that the leftmost M-type slot is assigned to the general M-unit, and that the leftmost I-type slot is assigned to the general I-unit. The lowercase "i" and "m" indicate that the instructions in the third and fourth slots are assigned to the restricted execution units: i-unit and m-unit. Third, the three B-units are also not equal – although in practice, the only difference that usually arises is with regards to "brp" instructions. Fourth, there are some exceptions to the left-to-right rule, as indicated by the footnotes in the figure. These exceptions arise for the following three reasons: 1) Itanium always assigns F-unit instructions from the second bundle to the f-unit, 2) Itanium always assigns I-unit instructions in the 6th slot to the i-unit, and 3) Itanium always stalls after any bundle containing a B-type instruction, unless that instruction is a "brp" or a "nop".

A.2 Constructing the FSA for our 6-wide VLIW

We now proceed to describing the state machines that we derive for our 6-wide VLIW, using the approach of [1]. One of the key contributions of [1] was the observation that partitioning the FSA can substantially reduce the number of states in the FSA. This partitioning is accomplished along disjoint functional units. In our case the M and I units are not disjoint, because their are A-type instructions that may schedule on either of these units. All other execution units are disjoint, but found that it a single partition was sufficient to reduce the state tables to an efficient size. The M and I execution units form one FSA and all other units form the second FSA.

In Table A.1, the FSA for the M and I units is considered. Because the A-type instruction may execute on either an M or an I execution unit, the A and I units must belong to a single Finite State Machine (FSA). Table A.1, requires some explanation. In this table, the 30 rows correspond to the 30 states in the FSA. Each of the 30 states corresponds to a specific set of instructions resources that are already spoken for. Looking at the first column of this table, each state is given both a state number and a set of consumed resources. For example, the 29th row contains “29:(111000)” in the first column. The meaning of the parenthetical 6-digit number summarizes the resources used: the first digit indicates the number of “i” instructions, the second digit indicates the number of “I” instructions, the third digit indicates the number of “m” instructions, the fourth digit indicates the number of “M” instructions, the fifth digit indicates the number of “a” instructions, and the sixth digit indicates the number of “A” instructions. The heading of the first column succinctly summarizes this encoding information by the expression “#:(iImMaA)”. Therefore, state #29 corresponds to one restricted I-type, one unrestricted I-type, and one restricted M-type (i=1, I=1, m=1, M=0, a=0, A=0).

The next six columns in Table A.1, describe the edges of the FSA. If, from state #29, an M-type or an A-type instruction is added, we transition to state #30. All other instruction types are marked with dashes, because these instruction types cannot be scheduled on the same cycle as those already scheduled. For a particular row in Table A.1, the summation of all of the parenthetical digits for that state will tell us how many instructions have been scheduled. For instance, state #29 has three instructions scheduled (1+1+1+0+0+0 = 3). In this way, we can see that no state has more than 4 instructions scheduled (since the hardware has only four M and I execution units). In addition, since each next state is derived by adding a single instruction type to the current state, we can also see that the next state always has a parenthetical-digit-sum that is larger by 1 than that for the current state.

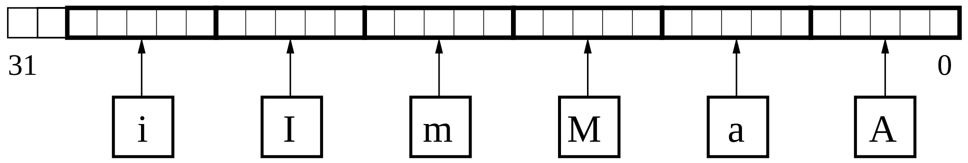


Figure A.7: Packing the next-state information from one row of Table A.1 into a 32-bit vector. With 30 states, each next-state value requires 5 bits (dashes become zeroes).

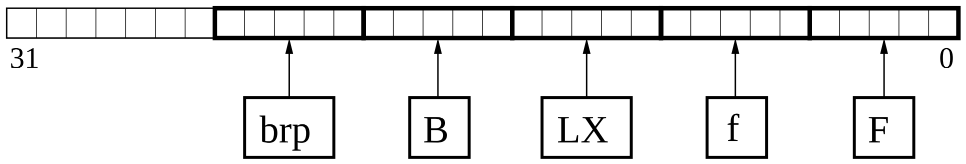


Figure A.8: Packing the information of one row from Table A.2 into a 32-bit vector.

Although there are no actual "A units" on our test machine, the A-type instructions must be maintained as part of the state, because the decision of whether to assign to a M or I unit may depend on later instructions. For instance, suppose that, from state #21 (i=1, I=0, m=0, M=0, a=0, A=1), an I-type instruction is added. Then, it would seem logical that the next state would be (i=1, I=1, m=0, M=0, a=0, A=1). But in fact, such a state does not exists, and Table A.1 indicates that the next state is #28 (i=1, I=1, m=0, M=1, a=0, A=0). This is because

Current State	Next State					
Case#:(iImMaA)	i	I	m	M	a	A
0:(000000)	6	5	4	3	2	1
1:(000001)	17	13	11	9	8	7
2:(000010)	19	14	19	10	19	8
3:(000100)	18	15	12	12	10	9
4:(001000)	19	16	-	12	19	11
5:(010000)	20	20	16	15	14	13
6:(100000)	-	20	19	18	19	17
7:(000002)	27	21	23	21	21	21
8:(000011)	25	21	25	21	25	21
9:(000101)	24	21	23	23	21	21
10:(000110)	26	21	26	26	26	21
11:(001001)	25	22	-	23	25	23
12:(001100)	26	23	-	-	26	23
13:(010001)	27	27	22	21	21	21
14:(010010)	28	28	28	21	28	21
15:(010100)	27	27	23	23	21	21
16:(011000)	28	28	-	23	28	22
17:(100001)	-	27	25	24	25	27
18:(100100)	-	27	26	26	26	24
19:(101000)	-	28	-	26	-	25
20:(110000)	-	-	28	27	28	27
21:(010110)	29	29	29	29	29	29
22:(011001)	29	29	-	29	29	29
23:(011100)	29	29	-	-	29	29
24:(100101)	-	29	29	29	29	29
25:(101001)	-	29	-	29	-	29
26:(101100)	-	29	-	-	-	29
27:(110100)	-	-	29	29	29	29
28:(111000)	-	-	-	29	-	29
29:(111100)	-	-	-	-	-	-

Table A.1: FSA States for the M and I units of our 6-wide VLIW.

Current State	Next State				
Case#:(bBLfF)	brp	B	L	f	F
0:(00000)	5	4	3	2	1
1:(00001)	14	10	7	6	6
2:(00010)	15	11	8	-	6
3:(00100)	16	12	9	8	7
4:(01000)	17	13	12	11	10
5:(10000)	17	17	16	15	14
6:(00011)	22	18	-	-	-
7:(00101)	23	19	-	-	-
8:(00110)	24	-	-	-	-
9:(00200)	-	-	-	-	-
10:(01001)	25	20	19	18	18
11:(01010)	26	20	-	-	18
12:(01100)	27	21	-	-	19
13:(02000)	28	28	21	20	20
14:(10001)	25	25	23	22	22
15:(10010)	26	26	24	-	22
16:(10100)	27	27	-	24	23
17:(11000)	28	28	27	26	25
18:(01011)	29	-	-	-	-
19:(01101)	-	-	-	-	-
20:(02010)	30	30	-	-	-
21:(02100)	31	31	-	-	-
22:(10011)	29	29	-	-	-
23:(10101)	-	-	-	-	-
24:(10110)	-	-	-	-	-
25:(11001)	30	30	-	29	29
26:(11010)	30	30	-	-	29
27:(11100)	31	31	-	-	-
28:(12000)	-	-	31	30	30
29:(11011)	-	-	-	-	-
30:(12010)	-	-	-	-	-
31:(12100)	-	-	-	-	-

Table A.2: FSA States for the F, LX and B units of our 6-wide VLIW.

the filling of both I-type execution units tells us that the A-type instruction must become an M-type instruction. In essence, the (i=1, I=1, m=0, M=0, a=0, A=1) has been merged into state #28.

This type of state merging is critical to achieving a small table. As new instructions types are added, prior instructions may move into more restrictive types. In the last paragraph, an A-type instruction became assigned to an M execution unit. Similarly, an M-type instruction may move to the restricted-M execution unit. For instance, from state #6 (i=0, I=0, m=0, M=1, a=0, A=0), a new M type instruction does not produce (i=0, I=0, m=0, M=2, a=0, A=0), but instead produces state #11 (i=0, I=0, m=1, M=1, a=0, A=0). Since two M-type instructions are needed, then one of them must be assigned to the more unrestricted execution unit, even though neither of these instructions specifically requires this unit.

Because this state table has only 30 rows, efficient data structures are possible. For instance, since 5 bits are needed to express a state number, and since there are 6 next states from the current state, the next-state transition edges will require a total of 30 bits, which fits within a single long integer.

In Table A.2, the FSA for the F, LX, B, and brp instructions is shown. The format of this table is analogous to Table A.1, except that there are only four next-state transitions per row. The brp instructions are separated from the general B instructions because the Itanium allows the brp instruction to be scheduled in some instances where other instructions cannot be (See footnote "d" on Figure A.6. Because the F and B units are disjoint, this FSA could be partitioned further. Since the the state table contains only 32 rows, however, it would be counter-productive to divide it further, because of the overhead of maintaining correctness across multiple FSAs.

Template limitations of the IA-64 test machine have reduced the number of states in Table A.2. For instance, state #8 in this table indicates two "LX" instructions. We notice that there is no next state for adding a "B" type instruction, even though all of the B units are idle. The B-type instruction cannot be added, because there is no template in Figure A.3 that contains two "LX" instructions along with a "B" instruction. Template restrictions were not an issue in the FSA of Table A.1, however, because the IA-64 provides more templates for M and I units than for other units.[1]

Our machine therefore requires 62 states ($30 + 32 = 62$). This is a far smaller number of states than the machine studied in [1] because there are no inter-cycle restrictions in our machine. Every resource is fully pipelined so that the instructions that are schedulable on a given cycle are not dependent on what had been scheduled on the previous cycle. But it must be remembered that a lack of inter-cycle *resource* dependencies is not to to say that there are no inter-cycle *data* dependencies. These dependencies are not the concern of the FSA, as it does not consider specific instructions, but only instruction types.

With two state tables, synchronization is required to maintain correctness. This issue arises because some state combinations exceed other, overall resource constraints, beyond those considered in the individual tables. For instance, in Table

[1]Examining Figure A.5, we see that 23 templates access both M units, and 16 templates use both I units. In contrast: 6 templates use all three B units, and 4 templates use both F units.

A.1, state #30 is a valid state that corresponds to using two M units and two I units. Similarly, in Table A.2, its state #30 is also valid, corresponding to the use of all three B units. Though these two states are individually legal, yet they are mutually exclusive – it is not possible to use two M units, two I units, and three B units at the same time, since this would require a 7-wide VLIW. Therefore, the VLIW width is one of the resource constraints that is not considered in either of these state tables. Another resource constraint arises from the templates available in our machine. For instance, state #20 in Table A.1, corresponds to the use of the two I units, while state #29 in Table A.2, indicates the use of 2 F-type instructions and 2 B-type instructions. These states are both valid, and the the total number of instructions between these two states is 6, which is also valid. Nonetheless, these two states are mutually exclusive, because, referring to the templates of Figure A.5, we observe that there is no template which permits 2 F-type instructions, 2 B-type instructions, and 2 I-type instructions. The solution is to create a new bit vector for each state in both tables, to indicate those templates that could be used to schedule the current resources. The AND-ing of these two state vectors will therefore indicates the set of templates that are able to schedule the instructions within *both* states. If this AND-ed vector is empty, then the states are mutually exclusive.

A.3 Constructing the FSA for our 3-wide VLIW

Table A.3 gives the FSA for our 3-wide VLIW. This FSA is much simpler than the one employed by our 6-wide VLIW. First of all, Table A.3 presents a unified table, where as the 6-wide VLIW is partitioned into two state machines (Tables A.1 and A.2).

A unified FSA is possible because the 38 states of Table A.3 are already sufficiently small to allow for efficient management. By avoiding partitioning, the overheads of verifying mutual compatibility among the partitioned states are also avoided.

There are two factors that cause the 3-wide FSA to be simpler. First, there are fewer instructions to consider. For state #29 in Table A.1 and states #29-#31 in Table A.2 all uses 4 instructions-issue slots, so they cannot happen on a 3-wide VLIW. Second, there are fewer resource constraints to consider – comparing Table A.3 with Tables A.1 and A.2, the "f" and "b" columns are absent from the 3-wide FSA. These absences are because the 3-wide VLIW has only the general F unit, so that restricted F-type instructions are not of concern, and because the unique rules for *brp*-instruction placement only apply to multiple-bundle long-words, as described in Section A.1.

The FSA shown in Table A.3 should be reduced further, in order to allow for an efficient representation. Although the FSA shown in Table A.3 is much simpler than the one presented in Table A.1 *together with* Table A.2, yet Table A.3 is more complex than either of these other two tables, *individually*; it has more rows and more columns than either of these other tables. We could, of course, choose to partition Table A.3 as we did for the 6-wide, producing two simpler tables. Since the full table only has 37 states, however, it is possible to create an efficient data structure through merging states. For example, state #15 and state #16 are both

Current State	Next State								
Case:(iImMaAFBL)	i	I	m	M	a	A	F	B	L
0:(000000000)	6	5	4	3	2	1	18	17	36
1:(000001000)	12	10	11	10	7	10	-	-	37
2:(000010000)	13	9	13	7	13	7	20	19	37
3:(000100000)	12	10	8	8	7	10	22	21	37
4:(001000000)	13	11	-	8	13	11	24	23	37
5:(010000000)	14	14	11	10	9	10	-	-	-
6:(100000000)	-	14	13	12	13	12	26	25	-
7:(000110000)	15	15	15	15	15	15	30	29	-
8:(001100000)	15	15	-	-	15	15	30	29	-
9:(010010000)	16	16	16	15	16	16	32	31	-
10:(010100000)	16	16	15	15	15	15	32	31	-
11:(011000000)	16	16	-	15	16	15	32	31	-
12:(100100000)	-	16	15	15	15	16	32	31	-
13:(101000000)	-	16	-	15	-	15	32	31	-
14:(110000000)	-	-	16	16	16	16	-	-	-
15:(101100000)	-	-	-	-	-	-	-	-	-
16:(111000000)	-	-	-	-	-	-	-	-	-
17:(000000010)	25	-	23	21	19	-	27	28	-
18:(000000100)	26	-	24	22	20	-	-	27	-
19:(000010010)	31	31	31	29	31	31	33	-	-
20:(000010100)	32	32	32	30	32	32	-	33	-
21:(000100010)	31	31	29	29	29	31	33	-	-
22:(000100100)	32	32	30	30	30	32	-	33	-
23:(001000010)	31	31	-	29	31	31	33	34	-
24:(001000100)	32	32	-	30	32	32	-	33	-
25:(100000010)	-	-	31	31	31	31	-	-	-
26:(100000100)	-	-	32	32	32	32	-	-	-
27:(000000110)	-	-	33	33	33	33	-	-	-
28:(000000020)	-	-	34	34	34	34	-	35	-
29:(001100010)	-	-	-	-	-	-	-	-	-
30:(001100100)	-	-	-	-	-	-	-	-	-
31:(101000010)	-	-	-	-	-	-	-	-	-
32:(101000100)	-	-	-	-	-	-	-	-	-
33:(001000110)	-	-	-	-	-	-	-	-	-
34:(001000020)	-	-	-	-	-	-	-	-	-
35:(000000030)	-	-	-	-	-	-	-	-	-
36:(000000001)	-	-	37	37	37	37	-	-	-
37:(001000001)	-	-	-	-	-	-	-	-	-

Table A.3: State table for our 3-wide VLIW. Unlike the 6-wide FSA, this is a unified table that includes all functional units.

final states – since both of these states already contain three instructions, there are no valid next-state transitions for either state #15 or state #16. Merging all of these final states into one state (new state #15), results in Table A.4. In Table A.4, only 28 states are needed. By merging all of the final states, their specific resource information is lost (as indicated by the dashes for state #15 of Table A.4), so that the state information can no longer be used to select a template, as it was for the states of the 6-wide FSA.

Examining Table A.4, we see that the merging of final states has in turn opened up new opportunities for merging; merging these states also results in the 19 states of Table A.5. For example states #7, #9, and #10 of Table A.4 all have equivalent next-state values, and so may be merged into state #7 in Table A.5. The original states corresponding to the new states of Table A.5 are indicated by the parenthetical list beside each of the state #.

Even with only 19 states, it is difficult to compact the next-state information of Table A.5 into a single 32-bit vector. One solution would be to use a 64-bit vector, but a 32-bit vector is sufficient, if offsets are used. As indicated in Table A.5, the next-state information may be divided into 4 regions. Because the data ranges within each region are smaller, they may be expressed in fewer bits, as illustrated in Figure A.9. The algorithm for identifying the next state from the current state is a simple matter of: 1) identifying the region that the current state belongs to, 2) identifying proper bits from Figure A.9, based on the edge type, and 3) adding the region's offset value to the next-state information bits from Figure A.9.

Current State	Next State								
Case:(iImMaAFBL)	i	I	m	M	a	A	F	B	L
0:(000000000)	6	5	4	3	2	1	17	16	28
1:(000001000)	12	10	11	10	7	10	-	-	15
2:(000010000)	13	9	13	7	13	7	19	18	15
3:(000100000)	12	10	8	8	7	10	21	20	15
4:(001000000)	13	11	-	8	13	11	23	22	15
5:(010000000)	14	14	11	10	9	10	-	-	-
6:(100000000)	-	14	13	12	13	12	25	24	-
7:(000110000)	15	15	15	15	15	15	15	15	-
8:(001100000)	15	15	-	-	15	15	15	15	-
9:(010010000)	15	15	15	15	15	15	15	15	-
10:(010100000)	15	15	15	15	15	15	15	15	-
11:(011000000)	15	15	-	15	15	15	15	15	-
12:(100100000)	-	15	15	15	15	15	15	15	-
13:(101000000)	-	15	-	15	-	15	15	15	-
14:(110000000)	-	-	15	15	15	15	-	-	-
15:(————)	-	-	-	-	-	-	-	-	-
16:(000000010)	24	-	22	20	18	-	26	27	-
17:(000000100)	25	-	23	21	19	-	-	26	-
18:(000010010)	15	15	15	15	15	15	15	-	-
19:(000010100)	15	15	15	15	15	15	-	15	-
20:(000100010)	15	15	15	15	15	15	15	-	-
21:(000100100)	15	15	15	15	15	15	-	15	-
22:(001000010)	15	15	-	15	15	15	15	15	-
23:(001000100)	15	15	-	15	15	15	-	15	-
24:(100000010)	-	-	15	15	15	15	-	-	-
25:(100000100)	-	-	15	15	15	15	-	-	-
26:(000000110)	-	-	15	15	15	15	-	-	-
27:(000000020)	-	-	15	15	15	15	-	15	-
28:(000000001)	-	-	15	15	15	15	-	-	-

Table A.4: A reduced state table for our 3-wide VLIW. Compared to Table A.3, this table has merged all final-states (*i.e.,* states using 3 instruction-issue slots) into a single state, #15. In the process of merging, state #15 loses its resource information, which is the reason for the dashes in its parenthetical-resource-descriptor. Without this resource information, the state no longer can be used to select a valid template for scheduling; for a 3-wide VLIW, however, the advantage of a reduced table size outweighs this disadvantage. After merging these final states, additional states become revealed as equivalent, such as states #9 and#10.

Current State		Next State								
Case:	(Merged states of Table A.3)	i	I	m	M	a	A	F	B	L
0	(0)	6	5	4	3	2	1	14	13	12
1	(1)	10	7	9	7	7	7	-	-	15
2	(2)	11	7	11	7	11	7	17	16	15
3	(3)	10	7	8	8	7	7	17	16	15
4	(4)	11	9	-	8	11	9	18	9	15
5	(5)	12	12	9	7	7	7	-	-	-
6	(6)	-	12	11	10	11	10	12	12	-
7	(7,9,10)	15	15	15	15	15	15	15	15	-
8	(8)	15	15	-	-	15	15	15	15	-
9	(11,23)	15	15	-	15	15	15	15	15	-
10	(12)	-	15	15	15	15	15	15	15	-
11	(13)	-	15	-	15	-	15	15	15	-
12	(14,25,26,27,36)	-	-	15	15	15	15	-	-	-
13	(17)	12	-	9	16	16	-	-	19	-
14	(18)	12	-	18	17	17	-	-	12	-
15	(15,16,29,30,31,32,33,34,35,37)	-	-	-	-	-	-	-	-	-
16	(19,21)	15	15	15	15	15	15	15	-	-
17	(20,22)	15	15	15	15	15	15	-	15	-
18	(24)	15	15	-	15	15	15	-	15	-
19	(28)	-	-	15	15	15	15	-	15	-

Table A.5: A reduced state table for our 3-wide VLIW. Triple lines are used to divide the state table into four regions, used for efficient representation, as described in the text. For each of the 19 states in this table, a parenthetical set indicates those states from Table A.3 that have been merged into each of the reduced states. Therefore, each of the original 37 states from Table A.3 may be found in exactly one of the parenthetical sets in this new state table.

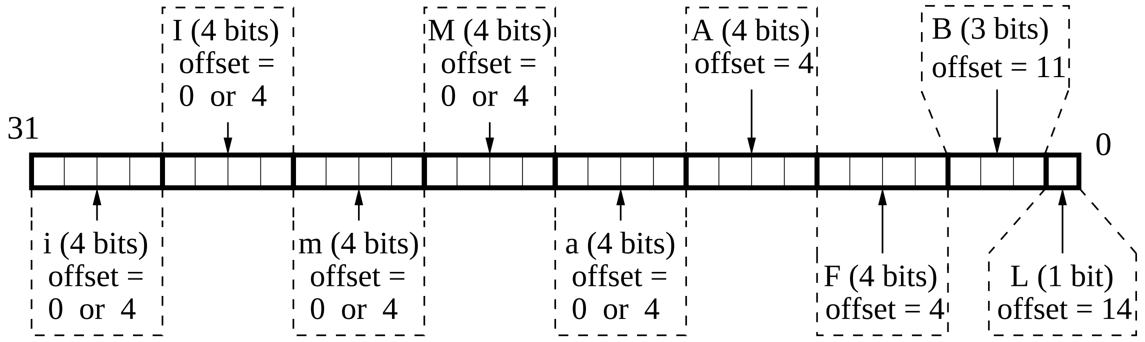

I (4 bits) offset = 0 or 4
M (4 bits) offset = 0 or 4
A (4 bits) offset = 4
B (3 bits) offset = 11
31
0
i (4 bits) offset = 0 or 4
m (4 bits) offset = 0 or 4
a (4 bits) offset = 0 or 4
F (4 bits) offset = 4
L (1 bit) offset = 14

Figure A.9: The packing of our 3-wide VLIW's next-state information into one long-word. An LX instruction from state # 0 is treated as a special case.

BIBLIOGRAPHY

[1] V. Bala and N. Rubin, *Efficient Instruction Scheduling Using Finite State Automata*", Proceeding of the 28th Annual International Symposium on Microarchitecture (MICRO-28)", November, 1995, IEEE Computer Society, Ann Arbor, Michigan, USA.

[2] J. Fisher, *Trace Scheduling: A Technique for Global Microcode Compaction*, ieeetc, July, 1981, Volume C-30, Number 7, 478-490.

[3] S. Freudenberger, T. Gross and P. Lowney", *Avoidance and Suppression of Compensation Code in a Trace Scheduling Compile*", . ACM Transactions on Programming Languages and Systems, July, 1994, Volume 16, Number 4, 1156-1214.

[4] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, . *The Superblock: An Effective Technique for VLIW and Superscalar Compilation*, The Journal of Supercomputing, January, 1993, Volume 7, Number 1, 229-248.

[5] P. H. Sweany and S. J. Beaty, *Dominator-path scheduling: a global scheduling method*, Proceedings of the 25th International Symposium on Microarchitecture, 1992.

[6] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran and W. Hwu, *Integrated Predicated and Speculative Execution in the*

IMPACT EPIC Architecture, Proceedings of the 25th International Symposium on Computer Architecture, July, 1998.

[7] P. Paulin, C. Liem, M. Cornero, F. Nacabal and G. Goossens, *Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends*, Invited paper, Proceedings of the IEEE, Volume 85, Number 3, March 1997.

[8] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli and Fred Homewood, *A Technology Platform for Customizable VLIW Embedded Processing*, Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, Canada, June 2000.

[9] S. Mahlke, D. Lin, W. Chen, R. Hank and R. Bringmann", *Effective Compiler Support for Predicated Execution Using the Hyperblock*, Proceedings of the 25th International Symposium on Microarchitecture, 1992.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*, (M. Kaufmann, San Francisco, CA, 1997).

[11] D. Kastner and S. Winkel, *IA-64 Instruction Scheduling ILP Dynamic*, Proceedings of the ACM SIGPLAN Workshop on Languages, Snowbird, Utah, USA, June 2001.

[12] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, . *Synthesis of embedded software from synchronous dataflow specifications*, The Journal of VLSI Signal Processing, June, 1999, Volume 21, Number 2, 151-166.

[13] P. Song, *Demystifying EPIC and IA-64*. Microprocessor Report, January, 1998, Volume 12, Number 1, 21.

[14] S. Haga and R. Barua, *EPIC Instruction Scheduling Based on Optimal Approaches*, First Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC01), Austin, TX, USA, December 2001.

[15] S. Y. Larin and T. M. Conte, *Compiler-driven cached code compression schemes for embedded ILP processors*, Proceedings of the 32nd Anual ACM/IEEE International Symposium on Microarchitecture, Haifa, Israel, November 1999.

[16] S. Aditya, S. Mahlke and B. R. Rau, *Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats*", . ACM Transactions on Design Automation of Electronic Systems, October, 2000, Volume 5, Number 4, 752-773.

[17] G. R. Beck, D. W. Yen and T. L. Anderson, . *The Cydra 5 Mini-supercomputer: Architecture and Implementation* , The Journal of Supercomputing, January, 1993, Volume 7, Number 1, 143-180.

[18] Texas Instruments, *TMS320C6000 Programmer's Guide*, Available at www.ti.com, 2003.

[19] Advanced RISC Machines Ltd. , *An Introduction to Thumb*, Mar. 1995.

[20] K. D. Kissell, *MIPS 16: High-Density MIPS for the Embedded Market*, Proceedings of Real Time Systems (RTS97), 1997.

[21] B. Rau and D. Yen and W. Yen and R. Towle, *The Cydra 5 Departmental Supercomputer*, IEEE Computer, January 1992, volume 22, number 1, 12-35.

[22] T. Zeitlhofer and B. Wess, *Code Optimization for the Carmel DSP-CORE*, Proceedings of the International Conference on Signal Processing Applications and Technology, Orlando, FL, November, 1999.

[23] S. Jee and K. Palaniappan, *Performance Evaluation for a Compressed-VLIW Processor*, ACM Symposium on Applied Computing, Madrdid, Spain 2002.

[24] Analogue Device, Inc, *The TigerSHARC Processor Family*, www.analog.com/processors/processors/tigersharc/.

[25] P. D'Arcy and S. Beach, *A New DSP Architecture for Portable Devices*, Wireless Symposium, Motorola, September 1999.

[26] Philips Corporation, *Philips Trimedia Processor Home Page*, www.semiconductors.philips.com/trimedia/.

[27] J. Bharadwaj, K. Menezes and C. McKinsey, *Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs*, Proceedings of the 32th International Symposium on Microarchitecture, Haifa, Israel, November, 1999, 262-271.

[28] S. Hanono and S. Devadas, *Instruction Selection, Resource Allocation and Scheduling in the* AVIV *Retargeting Code Generator*, Proceedings of the 35th ACM-IEEE Design Automation Conference, June 1998.

[29] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, *Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings*, Proceedings of the Annual International Symposium on Microarchitecture, 1996.

[30] H. C. Chou and C. P. Chung, *An Optimal Instruction Scheduler for Superscalar Processors*, IEEE Transactions on Parallel and Distributed Systems, March 1995, 303-313.

[31] K. Wilken, J. Liu and M. Heffernan, *Optimal Instruction Scheduling Using Integer Programing*, Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), 2000, 121-133.

[32] S. Liao, S. Devadas, K. Keutzer, S. Tjiang and Albert Wang, *Code Optimization Techniques for Embedded DSP Microprocessors*, Proceedings of the 32nd ACM-IEEE Design Automation Conference, 1995, 552-556.

[33] S. Liao, S. Devadas, K. Keutzer, *A Text-Compression-Based Method for Code Size Minimization in Embedded Systems*, ACM Transactions on Design Automation of Electronic Systems, January 1999, volume 4, number 1, 12-38e.

[34] J. Liu and F. Chow, *A Near-Optimal Instruction Scheduler for A Tightly Constrained, Variable Instruction Set Embedded Processor*, Proceedings of the 3rd ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Grenoble, France, October 2002.

[35] S. J. Beaty, *Genetic Algorithms and Instruction Scheduling*, Proceedings of the 24th Annual International Simposium on Microarchitecture (MICRO-24), Albuquerque, New Mexico, USA, November 1991, 206-211.

[36] S. Debray, W. Evans, R. Muth, and B. Sutter, *Compiler Techniques for Code Compaction*, ACM Transactions on Programming Languages and Systems, volume 22, number 2, March 2000, 378-415.

[37] S. Debray and W. Evans, *Profile Guided Code Compression*, Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), June 2002.

[38] J. Elder and M. Hill, *Dinero IV trace-driven uniprocessor cache simulator, release 7.* http://www.cs.wisc.edu/ markhill/DineroIV/, February, 1998.

[39] N. Stam, *Itanium 2: The Real IA-64 Deal, Inside Itanium 2.* Extreme Tech, July, 2002. http://www.extremetech.com/article2/0,1558,1152637,00.asp. (Academic Press, San Diego, CA, 2001), Chap. 1.

[40] J. Park and M. Schlansker, *On Predicated Execution.* Technical Report #HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May, 1991.

[41] R. Johnson and M. Schlansker, *Analysis Techniques for Predicated Code*, Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), IEEE Computer Society, Ann Arbor, Michigan, USA, December, 1996.

[42] J. R. Allen, K. Kennedy, C. Porterfield and J. Warren, *Conversion of Control Dependence to Data Dependence*, Proceedings of the 10th ACM Symposium on Principles of Programiing Languages, January, 1983, 177-189.

[43] A. E. Eichenberger and E. S. Davidson, *Register Allocation for Predicated Code*, Proceedings of the 28th ACM-IEEE International Symposium on Microarchitecture (MICRO-28), IEEE Computer Society, Ann Arbor, Michigan, USA, December, 1995.

[44] M. Smelyanskiy, S. Mahlke and E. Davidson, *Probabilistic Predicate-Aware Modulo Scheduling*, Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), March, 2004 173-179.

[45] J. Hoogerbrugge, L. Augusteijn, J. Trum and R. Van De Wiel, *A Code Compression System Based on Pipelined Interpreters*, Software Practice and Experience, Volume 42, Number 6, November 1998.

[46] S. Lucco, *Split-Stream Dictionary Program Compression*, Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), 2000, 27-34.

[47] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting, *Code Compression*, Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), 1997, 358-365.

[48] M. Franz, *Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-object Systems*, Mobile Ob-

ject Systems: Towards the Programmable Internet, LNCS Volume 1222, Fbruary 1997, 358-365.

[49] T. M. Kemp, R. M. Montoye, J. D. Palmer and D. J. Auerbach, *A Decompresion Core for PowerPC*, IBM Journal of Research and Development, Volume 42, Number 6, November 1998.

[50] C. Lefurgy, E. Piccininni and T. Mudge, *Reducing Code Size with Run-Time Decompresion*, Proceedings of the 5th International Symposium on High Performance Computer Architectures (HPCA 2000), January 2000, 218-227.

[51] J Lau, S Schoenmackers, T Sherwood and B Calder, *Reducing Code Size With Echo Instructions*, Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), October 2003, 84-94.

[52] K. D. Cooper and N. McIntosh, *Enhanced Code Compression for Embedded RISC Processors*, Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, May 1999, 139-149.

[53] H. Lekatsas and W. Wolf, *Random Access Decompression Using Binary Arithmetic Coding*, Data Compression Conference, March 1999, 306-315.

[54] Y. Xie, W. Wolf, H. Lekatsas, *Memory Hierarchies: A Code Decompression Architecture for VLIW Processors*", Proceeding of the 34th Annual Interna-

tional Symposium on Microarchitecture (MICRO-28)", December, 2001, IEEE Computer Society, Ann Arbor, Michigan, USA.

[55] M. Corliss, E. Lewis, A. Roth, *A DICE Implementation of Dynamic Code Decompression*", Proceeding of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems LCTES)", June, 2003.

[56] A. Suga and K. Matsunami, *Introducing the FR500 Embedded Microprocessor*, IEEE Micro, July, 2000, 21-27.

[57] E.A.Lee and D.G. Messerschmitt, *Synchronous Dataflow*, Proceedings of the IEEE, September 1997.

[58] D. August, J. Sias, J. Puiatti, S. Mahlke, D. Connors, K. Crozier, and W. Hwu, *The Program Decision Logic Approach to Predicated Execution*, Proceedings of the 26th International Symposium on Computer Architecture, Atlanta, GA, May, 1999.

[59] S. Horwitz, A. Demers, and T. Teitebaum, *An Efficient General Iterative Algorithm for Dataflow Analysis*, Acta Informatica, volume 24 number 6, November 1, 1987, 679-694.

[60] H. Sharangpani, K. Arora, *Itanium Processor Microarchitecture*, IEEE Micro, volume 20, number 5, September 2000, 24-43.

[61] C. Fraser, *An Instruction for Direct Interpretation of LZ77-compressed programs*, Microsoft Technical Report #MSR-TR-2002-90.

http://research.microsft.com/ cwfraser/papers/EchoTR.PDF, September, 2002.

[62] S. W. Golumb, *Shift Register Sequences*, (Aegean Park Press, revised edition, 1982).