UNIVERSITY OF CALIFORNIA, SAN DIEGO

Critical-Path Aware Processor Architectures

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Eric Tune

2004

Dedications

I thank my advisors, Dean Tullsen and Brad Calder, for their support, guidance, and inspiration. I thank all my labmates for listening and laughing when I pontificated. I thank Joyce Murphy, whose voice always relaxed me when my studies became stressful. Most significantly, I thank my mother, father and brother for their patience and support through this process.

It is easier to be critical than correct. – Benjamin Disraeli

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Portions of Chapters IV and V reproduce material which appears in the proceedings of the Seventh International Symposium on High-Performance Computer Architecture. The dissertation author was the primary researcher and author and the co-authors involved in the publication directed, supervised, and assisted in the research which forms the basis for that material.

The text of Chapter VI is in part a reprint of the material as it appears in the proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques. The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter VI.

The text of Chapter VII is in part a reprint of the material as it appears in the proceedings of the 37th International Symposium on Microarchitecture. The dissertation author was the primary researcher and author and the co-authors involved in the submission directed or assisted the research which forms the basis for Chapter VII.

VITA

1974	Born, Orange, California
1992	High School Diploma Sacramento, California
1996	B.S. in Computer Engineering University of California, San Diego
2000	Internship, Compaq Computer Corporation, VSSAD Group, Shrewsbury, Massachusetts
2001	M.S. in Computer Engineering University of California, San Diego
2004	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Eric Tune, Rakesh Kumar, Dean Tullsen, and Brad Calder "Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy", To appear in the proceedings of the *37th International Symposium on Microarchitecture* (MICRO 2004), December 2004, Portland, OR.

Eric Tune, Dean Tullsen, and Brad Calder. Quantifying Instruction Criticality, In the proceedings of the *11th International Conference on Parallel Architectures and Compilation Techniques* (PACT 2002), September 2002. Charlottesville, VA

Eric Borch, Eric Tune, Bobbie Manne, and Joel Emer. Loose Loops Sink Chips *In the proceedings of the Eighth Int'l Symposium on High-Performance Computer Architecture*, (HPCA 2002), February 2002, Cambridge, MA

John Seng, Eric Tune, Dean Tullsen, and George Cai. *Reducing Processor Power with Critical Path Prediction In the proceedings of the 34th International Symposium on Microarchitecture* (MICRO 2001) December 2001, Austin, TX

Eric Tune, Dongning Liang, Dean Tullsen, and Brad Calder. Dynamic Prediction of the Critical Performance Path, *In the proceedings of the Seventh International Symposium on High-Performance Computer Architecture* (HPCA 2001) January 2001, Monterrey, Mexico

ABSTRACT OF THE DISSERTATION

Critical-Path Aware Processor Architectures

by

Eric Tune
Doctor of Philosophy in Computer Science (Computer Engineering)
University of California, San Diego, 2004
Professors Dean Tullsen and Brad Calder, Co-Chairs

Modern processors remove many artificial constraints on instruction ordering, permitting multiple instructions to be executed in parallel. As a result, only a fraction of all the instructions in a program trace determine the execution time of the program. Any effort to improve program performance is wasted when not applied to these *critical* instructions. Likewise, the remaining *non-critical* instructions may be delayed, to a point, without affecting performance. Depending on the program and microarchitecture, typically between a few percent and half of all dynamic instructions are critical. We propose and evaluate several hardware techniques to classify whether an instruction is critical or non-critical, and discuss related efforts at the same. We show that the criticality of dynamic instructions is correlated to the corresponding static instruction. We exploit this correlation to predict an instruction's criticality, in hardware, before it executes. We call this *critical-path prediction.* These predictions can be used anywhere that the processor must arbitrate between instructions for a limited resource. We demonstrate the utility of these predictions in several such applications, which we call *critical-path aware optimizations*: a processor with a limited-rate value-predictor, a clustered microarchitecture with inter-cluster communication delays, and a reduced-power microarchitecture with heterogeneous functional units and queues. We perform an offline analysis of the critical paths of programs to validate our findings and to quantify the degree of criticality of different instructions. Our findings lead us to propose a new multithreading architecture. Under our proposal, threads execute in parallel in a manner sensitive to the hardware implications of supporting multiple contexts, and also sensitive to the critical path issues. We use execution-driven simulation to evaluate the performance of all the processor designs which we propose.

I

Introduction

Improvements in microprocessor performance have far-reaching benefits. High-performance microprocessors are the central component in the computer systems that process financial transactions, predict the weather, aid the development of new medicines, and sit on 600 million desktops worldwide[1]. And, as semiconductor manufacturing technologies have improved, architectures which were once reserved for high-performance processors are applied to embedded processors, of which there are billions[94]. Therefore, performance improvement for microprocessors is an important area of inquiry.

The tremendous improvements in microprocessor performance are largely due to improvements to semiconductor fabrication technology, which result in larger transistor counts and faster switching. The increasing transistor counts have permitted the design of complex processors which allow many instructions to be active in the pipeline concurrently.

Although miniaturization has carried microprocessor performance a great way, new factors threaten to limit gains from further miniaturization. The number of fast circuit blocks is limited by power constraints. The growing delay of wires, relative to transistors, limits the complexity of structures that can be accessed in a clock cycle. This constrains the size of and number of ports on caches, prediction tables, and register files. Further innovation is needed at the architectural level to work around these resource constraints.

Together, high concurrency and multiple resource constraints mean that a microprocessor has to make decisions about how resources are allocated among individual instructions on a cycle by cycle basis. However, a comprehensive mechanism for making these decisions has been largely lacking. In this thesis, we develop mechanisms for determining which instructions should get access to limited resources, in order to maximize performance, based on the *criticality* of instructions. We propose mechanisms for identifying which instructions lie on the *critical path* of executing a program, and show that considerable improvements in performance result when *critical* instructions are given priority to use constrained resources. We relate our findings to another microarchitectural technique which increases concurrency: multithreading. We propose a new multithreading technique which exploits portions of a programs critical path which are dominated by a single instruction.

In the broadest terms, the goal of this research is to find ways to improve the performance of a microprocessor by intelligently managing its resources.

I.A The Critical Path

The idea of a critical path has been used in planning and operations research for many years. Most projects can be thought of as consisting of a set of tasks, each with some duration, and a set of constraints on the order in which tasks can be performed. The longest ordered sequence of tasks determines the minimum time to complete the project. This set of tasks, linked by constraints, constitutes the critical path.

Consider as an example, building a house. Several tasks comprise this project: framing the walls, plumbing, wiring, roofing, and plastering the walls

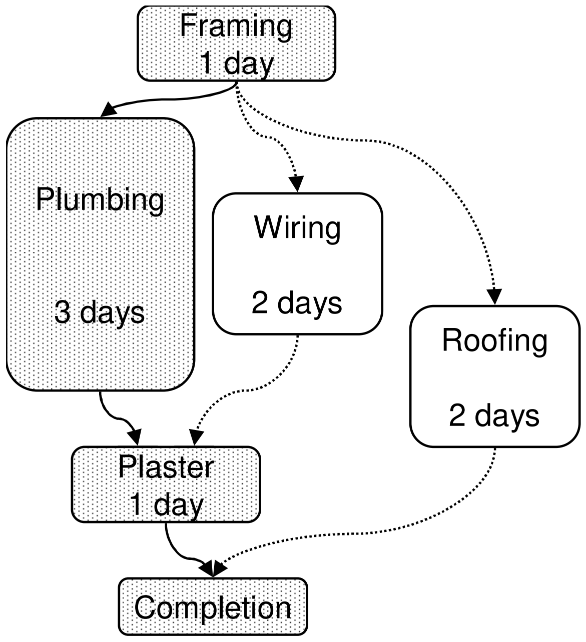


Figure I.1: Example of a critical path in a house construction project.

Figure I.1 illustrates the constraints on the ordering of those tasks. For example, the walls cannot be plastered until the wiring and plumbing has been completed, as indicated by edges between the boxes in the figure. The longest sequence of events consists of *framing*, then *plumbing*, then *plastering*. This is the critical path, and is highlighted.

A general contractor planning the construction of the house may wish to know how he can finish the house sooner, or how he can finish the house in the same amount of time, at a lower cost. If he is building several houses in several locations, he may wish to know how to allocate his workers to different projects. Knowledge of the critical path provides a starting point from which to answer these questions. Consider two important properties of the critical path. First, the duration of the project can only be reduced by shortening a task on the critical path. Second, a task not on the critical path can be made to take longer, up to a point, without increasing the duration of the project. These two observations address the contractors two questions. In the context of this example, he might wish to hire a faster, perhaps more expensive plumber, or he may choose to hire slower, but less expensive electricians and roofers. In short, knowing what tasks lie on the critical path provides a starting point for making decisions which trade off the duration of a project with other costs.

An instruction trace of a program running on a microprocessor can be thought of as a project as well. Dynamic instructions correspond to tasks. Data dependencies are one type of constraint on the ordering of instructions. The processor hardware also puts additional constraints on the scheduling of instructions. The critical path of a graph of these constraints represents the runtime of the program. Like the contractor in the example, a processor architect wants to minimize program duration, and minimize other costs. Those other costs may include power consumption, or may relate to wire delay, which, as mentioned earlier, are growing concerns. We refer to those instructions which lie on this critical path as *critical instructions*, and the rest as *non-critical instructions*.

However, a general-purpose processor must be designed to run many different kinds of programs, thus a processor architect cannot, or should not, design a general purpose processor with regard to the critical path of a single program. Instead, this thesis proposes incorporating additional hardware into microprocessors which would:

- **Identify** which instructions are limiting performance based on their behavior as they pass through the pipeline, and their dependencies with other instructions.
- **Predict** which instructions will be critical as they enter the pipeline, based on which instructions were previously identified as critical.
- **Control** the behavior of the processor at the microarchitectural[1] level using

[1]The *microarchitectural level* is defined, for this thesis, as high-level aspects of a processor design apart from

those predictions.

We refer to processor designs which implement these three features as *Critical-Path Aware Processor Architectures.*

Each of these steps is performed continuously in the processor, and each presents challenges. Although it is possible to analyze the critical path of a program after it is run, (and we do so), it is much more useful to be able to identify the critical path as the program runs. In order to control the behavior of the processor, it is necessary to know whether a dynamic instruction[2] is critical as soon as it enters the pipeline. But, its criticality depends on what happens when and after it executes, making it infeasible to say with certainty whether an instruction is critical or non-critical when it first enters the processor. However, we observe if a recent instance of a static instruction was critical, then subsequent instances will also likely be critical. This allows us to observe whether instructions were critical after they execute, to record this behavior, and then to predict that future instances of the instruction will have the same status. We refer to the process of identifying and predicting potentially critical instructions as *Critical-Path Prediction.*

Despite the fact that there is some uncertainty in both the identification and prediction steps, we find, in our evaluations, that the information produced by critical-path prediction can both improve performance over previously proposed policies which do not use critical-path predictions, and enables new microarchitectures which would not have been sensible without critical-path information. We refer to the act of using predictions to control the microarchitecture as *Critical-Path Aware Optimization.*

the instruction set architecture.

[2]A *dynamic instruction* is a single instance in the instruction trace of a program, as distinct from a *static instruction*, which is a sequence of bytes in the executable file of a program.

I.A.1 Why now?

Critical-path analysis has been used formally for at least 35 years. Why has it not been applied to microprocessors before, in the way we propose? To be sure, critical-path analysis is used extensively in circuit design. A circuit designer needs to know which paths through a circuit have the greatest delay, in order to make the entire circuit block faster, and to set a clock period. Critical path analysis has also been performed in compilers to schedule instructions. This, and other, software-based approaches to critical path analysis of programs, are discussed in section III. Here we distinguish between a circuit critical path and a program critical path. This distinction may be less clear in the domain of application-specific processors [43] and reconfigurable processors [46], where the program and the hardware are designed at the same time. In that domain, a critical path analysis might include both software and hardware. However, in this thesis, we are concerned with programs running on general-purpose processors, where the processor designer cannot make program-specific design choices.

Why, then, had the critical path concept not been applied to analyzing the performance of individual instruction in a processor, prior to the work which forms the basis for this thesis? We suggest some contributing factors. First, an increases in the number of transistors on a typical processor die mean a reduction in the cost of the additional hardware needed to implement critical-path prediction. These additional transistors can also be used to implement circuit blocks that offer multiple levels of service. Knowing which instructions are critical or non-critical is not useful without having a way to offer them different levels of service. In several cases, these multiple levels of service stem from the constraints of wire delay and power consumption. As mentioned previously, these constraints are recently matters of particular concern in processor design.

Second, contemporary processor designs have removed many of the constraints on the order in which instructions are executed which would otherwise largely serialize the execution of programs, leaving all instructions on the critical path. Knowing that all instructions are critical is hardly useful. This point deserves particular attention. In a scalar, in-order processor, all instructions are on the critical path. Speeding up any instruction would make the program run faster, albeit perhaps by only one clock cycle. It is only when many instructions are not critical, that it is useful to know which ones are. As transistor budgets have increased, each generation of processor has removed more constraints on the order in which instructions can execute. Register renaming and out-of-order execution, [6, 81, 104], remove artificial dependences imposed by instruction ordering. Branch prediction [101] can remove many control dependencies. When these constraints are removed, the critical path may consist of only a fraction of the instructions in the program.

In our work, we find that, typically, between 5% and 40% percent of instructions are critical. This number varies considerably depending on the processor design and the program. And, with each generation of processors removing more artificial constraints on the ordering of instructions, the number of critical instructions will decrease further, and the relative importance of each critical instruction will increase. The implication is significant: *It does not make sense to treat all instructions the same.*

I.B Overview of Dissertation

Chapter II provides background information on several topics which relate to this thesis.

Chapter IV describes *Critical Path Prediction* for dynamically identifying instructions likely to be on the critical path. We propose several critical-path predictors We also describe critical-path predictors and related schemes proposed by other researchers

Critical path prediction does not, by itself, make programs run faster. Instead, it is a *meta-optimization.* It provides information that can be used to control other hardware optimizations, making them more effective than they would be without critical path predictions. Chapter V describes our evaluation of three critical-path aware optimizations, Sections V.A and V.B evaluate how critical path predictions can be used to control value prediction and clustered architecture instruction steering. We show that critical path prediction has the potential to increase the effectiveness of these hardware optimizations by as much as 70%, without adding greatly to their cost. Section V.C evaluates how critical path information can be used to reduce power consumption, while minimizing performance losses from power constraints. We also describe related work on, and suggest new kinds of critical-path aware optimizations.

Chapters IV and V firmly establish the feasibility of analyzing the critical path of a program at runtime, and using that analysis to direct hardware-level performance improvements. Chapter VI seeks to deepen our understanding of the critical paths of programs. In that chapter, we describe an offline analysis of programs traces which we performed to better understand certain properties of critical paths. This analysis allows us to (i) study how the criticality of instructions changes (ii) measure the accuracy of proposed critical path predictors, (iii) quantify the amount of slack present in non-critical instructions, and (iv) provide a new metric, called tautness, which ranks critical instructions by their dominance on the critical path, and in time.

Our studies lead us to investigate multithreading, and to propose a new multithreading architecture. Chapter VII describes this new multithreading architecture, which does not use critical path prediction, but which builds on understanding gained from our work on the critical path, and which is improved when we apply a critical-path-related control policy.

II

Background

This chapter provides background information on topics which relate to this thesis. Section II.A explains value prediction, a microarchitectural optimization which we study in conjunction with critical path prediction in Section V.A. Section II.B provides some background on the importance of power consumption as a design constraint for microprocessors. We consider critical-path aware power reductions in Section V.C Section II.C provides some background on multithreading. Many readers may find this material familiar.

II.A Value Prediction

In Section V.A, we study how critical path prediction can be used to control *value prediction*, [69, 71, 39]. Value prediction exploits *value locality* in programs: the same static instructions often produce the same value, one of a small set of values, or a value which differs only by a fixed increment between subsequent dynamic instances. Thus, the result of an instruction may be predicted before it actually executes. In modern processors with large instruction queues, they may be many instructions which have been dispatched, but which are waiting for their operands to become ready. By predicting the outcome of

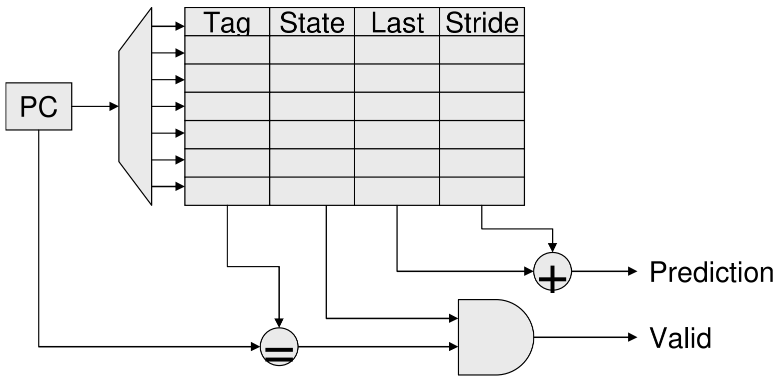


Figure II.1: A Stride Value Prediction Table.

an instruction, dependent instructions can execute before their operands would otherwise be ready. The predicted outcome is eventually confirmed. In event of a misprediction, at least the instructions affected by the incorrect prediction must be re-executed. Value prediction is of particular interest as a critical-path aware optimization. Unlike optimizations which merely reduce the latency of an instruction, value prediction *removes* data-dependencies altogether. Thus, there is the potential to drastically reduce the critical path if value prediction is applied to a select few instructions.

To implement value prediction, a conventional architecture is extended to include a value prediction table, and facilities for using, verifying, and recovering from the misuse of speculative values. The value prediction table tracks the results of completed instructions, identifies patterns, and predicts the outcome of future instructions. An example of one type of value prediction table, a *stride value prediction table*, is shown in Figure II.1. The value prediction table is a small RAM, like a cache. The table is indexed by the program counter (PC) of the instruction to be predicted. Each line of the table is a predictor for one static instruction. Each line of the table contains i) a partial tag, to aid in detecting aliasing, ii) predictor state, which specifies whether the predictions contained in this line are valid, and optionally, a confidence value, which is simply a count of the number of recent correct predictions, iii) the last value produced by an instance of the instruction predicted by this line, iv) a stride, which is the difference between the last value and the value before that. To make a prediction, the program counter is used as an index into the table. If the tag matches, and if the predictor is valid and confident, then the prediction is deemed valid. The sum of the stride and the last value is the prediction for the new value.

Completed instructions are used to update the prediction table. (The update path is not shown in the figure.) The predicted value is used in speculative computations, which are eventually committed once the predicted instruction is actually executed and the prediction is confirmed.

More complex value prediction schemes can predict other patterns of value locality. In our experiments in Section V.A, we use a hybrid value predictor [123], which, in addition to making stride-based value predictions as described above, can also make context-value predictions. The context value predictor predicts that the next result of an instruction will be one of the 4 most recently produced results, based on the history of the previous results. We do not evaluate instruction reuse [103], which is similar to value prediction.

II.B Power

Power consumption is an important design consideration for microprocessors. Design goals for a high-performance microprocessor systems include maximizing performance within limits for whole-chip average and peak power consumption, and avoiding areas of high power-density. Two components of power consumption are dynamic and static power consumption. The dynamic power of a block of logic is the power which is dissipated as a result of logic gates switching. Static power is power which is dissipated independent of any switching activity. Both are increasing in high-performance microprocessors [55], but static power consumption is increasing more rapidly [61], both in absolute terms, and in proportion to dynamic power consumption.

A block of logic in a processor may be implemented in several ways, each with different power and performance characteristics. A designer can vary device parameters, such as width, supply voltage and threshold voltage; the style of logic used, such as static CMOS or domino logic; or the high-level circuit design of a block, such as a carry-lookahead or carry-select adder. For purposes of discussion, we say that a logic block admits to a *fast/hot* or *slow/cool* implementation.

The simplest approach to meeting power constraints is for a designer to select one of several possible implementations for each block, such that the performance of the processor is maximized, and power constraints are not exceeded under any circumstances. Rather than designing for the worst case, the designer can opt to design for the average case, and provide a mechanism to reduce the speed of the entire processor, or of sections of the processor, when power consumption or chip temperature exceeds a threshold [83, 82, 14, 52].

Scaling the speed of the whole processor affects all instructions. However, not all instructions have the same performance needs. A better combined power/performance characteristic can be achieved if power decisions are made on a per-instruction basis. To achieve this, some blocks of a processor are implemented in both fast/hot and slow/cool styles. In Section V.C, we discuss how critical-path prediction can be used to steer instructions to either a slow/cool or fast/hot execution path, according to each instruction's criticality. One obvious place to apply this technique is a *functional unit*: a block which performs the arithmetic or logical operations specified by an instruction. We use the term *multi-speed* or *heterogeneous functional units* to refer to designs where the same arithmetic or logical function is implemented in both fast/hot and slow/cool styles.

The Intel Pentium 4 microprocessor has fast and slow integer functional units [49]. The slow functional unit runs at the core processor clock rate and the fast functional unit runs at twice that speed. Although this design highlights the feasibility of *multi-speed functional units*, the designers may have chosen it for reasons other than power savings. The fast functional unit only performs simple operations, such as addition and subtraction, while the slow functional unit performs all types of integer operations. The mechanism used to schedule instructions on this processor is not published.

Typically in a high performance processor, a functional unit is implemented to complete basic operations like addition and subtraction in a single cycle. When this constraint is relaxed, both the number and size of transistors may be reduced[11]. Both static and dynamic power is reduced when fewer and narrower (thus slower) transistors are used. An especially effective way to reduce static power, when the underlying technology permits it, is to raise the threshold voltage, which reduces the switching speed somewhat, but exponentially reduces sub-threshold leakage current [17], a major component of static power dissipation. Functional unit power may also be reduced in other ways, such as reducing power used to compute on operands with fewer significant bits[13]. Speculation control is another important microarchitectural technique for power reduction [74, 44].

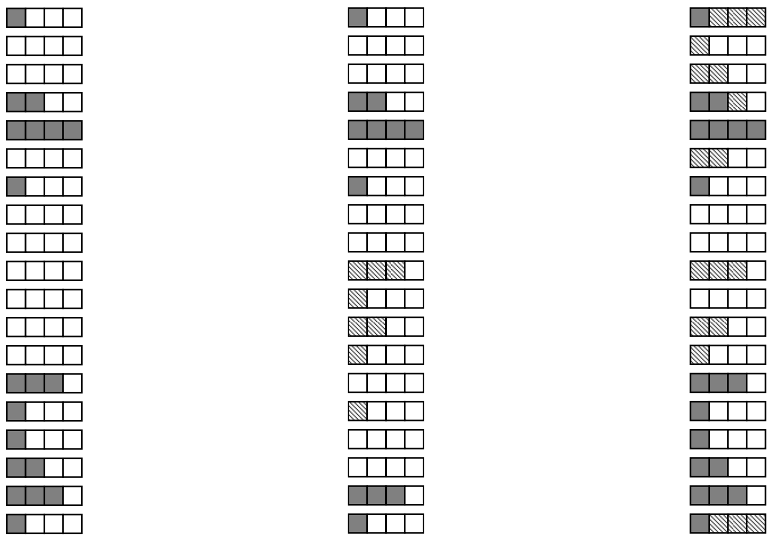
In addition to the work of Seng et al. which we discuss in Section V.C, several other researchers have also evaluated multi-speed functional units. Casmira and Grunwald [20] measured how much slack was present in instruction sequences, in order to determine how often slow functional units could be put to use. Pyreddy et al. [86], also studied a design with multi-speed functional units, using a static critical path prediction, based on the heuristics presented in Chapter IV. Semeraro et al. [95], performed an offline analysis of slack to independently control clock speeds in different blocks of a processor, but their technique did not control on a per-instruction basis. Fields, et al. also used their critical path predictor to control a pipeline with multi-speed functional units. That work is discussed in Section V.D.

Sato, et al. [91], reevaluated the results of Seng, et al., [97]. They found multi-speed functional units to be useful in decreasing the energy-delay product, but questioned the benefit of the split instruction queue design. They suggest that energy savings would be greater with a more accurate critical path predictor.

In Section V.C, we also discuss power-reduction for the instruction queue. Out-of-order execution allows ready instructions to be executed sooner than then they appear in program order. Because of control dependencies, and variable instruction latencies, the schedule produced by the compiler is often worse than that which can be produced by a hardware scheduling mechanism. In recent processors, that scheduling mechanism is provided by an *instruction queue*. Reduction of power consumption for the instruction queue has been the focus of some research [38, 18]. The instruction queue serves two purposes: to serve as a buffer between the fetching and execution portions of the pipeline, and as a mechanism for scheduling instructions. If the number of instructions that are considered for scheduling each cycle is reduced, then the power consumed by the instruction queue will likewise be reduced.

II.C Multithreading

Multithreading can refer to a class of hardware techniques which make better use of a processor by executing instructions from different threads without software intervention. Figure II.2 illustrates the time-density of instruction execution for a conventional processor and for two types of multithreading processors. The vertical dimension represents the passage of time over the span of 19 clock cycles. Each row of 4 boxes represents a capability to issue up to 4



(a) Single thread processor (b) Coarse-grain multi-threading processor (c) Simultaneous multi-threading processor

Figure II.2: Illustration of instruction throughput in single thread, coarse-grained multithreading, and simultaneous multithreading processors.

instructions per cycle. The shaded boxes represent an instruction from one of two threads executing on that cycle. The shaded boxes represent an instruction executing on that cycle. The white boxes represent an opportunity to make better use of the processor. Figure II.2(a) shows that a single-threaded processor may often be idle for short or long periods. These *stalls* are caused by events such as cache misses and branch mispredictions. Accesses to main memory can't even be shown to scale. Figure II.2(b) illustrates *Coarse-grained multithreading* (CGMT) [3, 90, 108, 78, 12]. When a cache miss occurs in one thread, execution is switched to a second thread after a delay. In this figure, instructions from two different threads are shown, and two thread swaps occur, with two cycles between each. In practice, the delays could be greater.

A *Simultaneous multithreading* (SMT) [115, 112, 50, 126] processor is capable of issuing instructions from 2 or more threads on the same cycle. This additional flexibility allows for a higher utilization of the processors execution units, as illustrated in Figure II.2(c). Another multithreading mode, not illustrated here, is Fine-grained multithreading [100, 5, 45, 67], whereby the processor executes instructions from a different thread each cycle, typically in a round-robin fashion.

Both forms of multithreading are attractive, compared to simply having multiple independent processors. Compare a multi-threaded processor and two separate single-thread processors. If both configurations are capable of the same throughput, the multithreaded processor will likely require much less die area, since only some logic is duplicated to support multiple thread. Further, in those cases when there is only a single thread to run, that single thread has the potential to have a higher instruction throughput on the multithreaded processor, than on one of the less capable single processors.

In these illustrations, only two threads are shown. However, it may often be desirable to build processors which support more than 2 threads. Trends in the characteristics of main memory (DRAM) make additional threads even more attractive. While DRAM memory latencies have grown relative to processor clock speeds [125], memory bandwidths have increased steadily. Multithreading offers a good way to exploit additional memory bandwidth with a single processing core.

However, support for additional threads has a cost. The state for the additional threads increases the size of the register file, and possibly other structures. This in turn can affect processor clock speed or pipeline length. In Chapter VII, we consider ways to support additional threads while avoiding these undesirable consequences.

III

Critical Path Analysis

In this chapter, we give some background on general critical path analysis, and we describe several ways in which critical path analysis can be applied to programs. We classify critical path analyses of programs into three categories: Static analysis is performed on machine code before run-time, but may make use of execution profiles. Offline analysis is performed on an instruction trace, after runtime. Dynamic analysis is performed as the program is run.

III.A General Critical-Path Analysis

The concept of a *critical path* was formalized by operations researchers, in the context of planning engineering and construction projects [58, 76, 57, 9]. A project may be represented by a directed acyclic graph. All project graphs have *source* and *sink* nodes which represent the initiation and completion of the project, respectively. There are two common ways of representing a project with a graph: *activity-on-arc* (AOA), and *activity-on-node* (AON). In the AOA format, each edge (or arc) represents an activity, and a weight on an edge indicates the time to complete the activity. Nodes represent discrete events. Under this format, it is typically necessary to either use zero-weight edges or for the same activity to be represented on several nodes. This is the format used in the graphical model proposed by Fields et al. and described in Section III.E. In the AON format, nodes represent activities, with the weight on each node representing the duration of the activity. Unweighted edges constrain the ordering of events. This is the format used in several diagrams in this thesis.

Regardless of the format used, the *earliest finish time* of a project is the length of a longest path through the project's graph. This is the critical path. In some cases, there can be multiple parallel sub-paths of equal length. The difference between the earliest time at which an activity could be started and the latest time at which it could be started without delaying the finish time, is termed the slack of that activity. More precisely it is termed the "total float"[76] (which Fields et al. [32] refer to as "global slack"). Critical edges have no slack. Delaying the initiation of any critical activity will delay the completion of the entire project.

Critical-path analysis has been used in several other areas of computer engineering. Critical path analysis has been used to analyze the performance of parallel programs, [127, 4, 68], and is of great importance in the design and layout of digital circuits, [107].

III.B Static Critical Path Analysis of Programs

A compiler may reorder instructions within a basic block. For an *in-order processor*, also called a *statically scheduled* processor, instructions within a basic block are executed in the order they appear in the program text, although a *superscalar* processor may issue and execute multiple consecutive instructions in a single cycle. The ordering of instructions emitted by the compiler may have a considerable effect on the resulting program's performance. If a directed graph is created to represent the instructions in a basic block and their data dependencies, then any topological sort of that graph is a valid ordering of the program text. Gibbons and Muchnick [41] describe an efficient algorithm to schedule (select an order for) the instructions of a basic block. Their algorithm tries to avoid *pipeline interlocks*, which are cycles in which the next instruction cannot execute because its dependencies are not satisfied. At each step, their algorithm uses one of several heuristics to select the next instruction to emit. One of those heuristics is to prefer instructions which lie on the longest path through the graph of unscheduled instructions. In this manner, their algorithm takes into account the critical path within a single basic block.

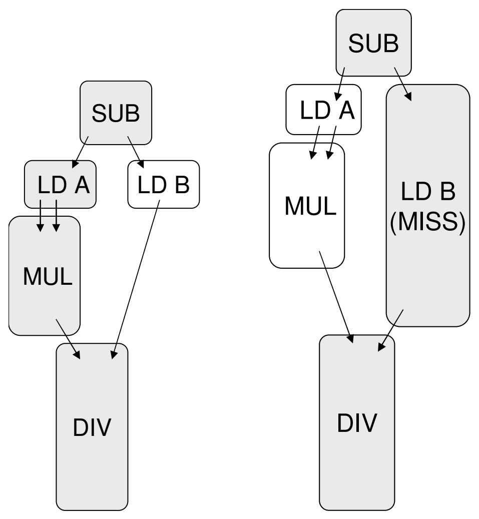
An instruction scheduler is limited in its ability to reorder instructions by the small size of many basic blocks. Fisher describes how to combine multiple basic blocks into larger traces [36]. Likely control flow paths can be determined by compile time heuristics, as in [73] or profiling [124]. Fisher's scheduler gives priority to instructions which lie on the longest path through the graph of unscheduled instructions. Lowney et al. [73] discuss the importance of considering the critical path in a trace scheduling compiler. Schlansker and Kathail [93] describe compiler transformations which reduce the height of the critical path through a *superblock*. A superblock, [53], is a set of basic blocks combined using predication.

The compiler analyses described above are associated with statically scheduled processors. For an *out-of-order* processor, also called a *dynamically scheduled* processor, the ordering of instructions within a program trace instructions will have much less of an effect on performance. *This is the type of processor we concerned with in this thesis.* Two alternative instruction orderings, which differ in performance when run on an in-order processor, may be be scheduled in the same order by a dynamically scheduled processor, and thus give the same performance there. When coupled with control speculation, dynamic scheduling can have a similar effect to trace scheduling, in that instructions are reordered across basic block boundaries.

Instruction scheduling is only one area where awareness of the critical path can be put to use. In this thesis, we consider how with how instructions can be classified as critical or non-critical, and how that classification can be used to guide many processor policies, and to enable new microarchitectures. A static approach, similar to instruction scheduling, could be used to classify instructions as critical and non-critical. However, any static analysis will have several shortcomings: i) it either assumes a particular control path or cannot properly account for data dependencies between basic blocks, ii) it assumes to fixed latencies for variable-latency instructions, such as loads, iii) it requires recompiling or at least reprocessing object code, iv) it assumes features of a particular processor model, or ignores microarchitectural effects on execution, and v) the ISA would likely have to change to convey critical/non-critical classifications[1].

The first two points each deserve an illustration. Figure III.1 illustrates the effect of variable instruction latency on the critical path. Figures III.1(a) and III.1(b) illustrate a data-flow graph of a fragment of code with 5 instructions. The height of each box is a metaphor for the latency of the instruction. The shaded boxes show instructions on the critical path for this fragment. Suppose, for the sake of the example, that the `div` instruction is definitely on the critical path. When both loads hit in cache, the first load and the multiply instruction are on the critical path, and the second load instruction is not critical. When the second load instruction misses in cache, the opposite is true. A static analysis which incorrectly estimated the latencies of the loads would incorrectly classify

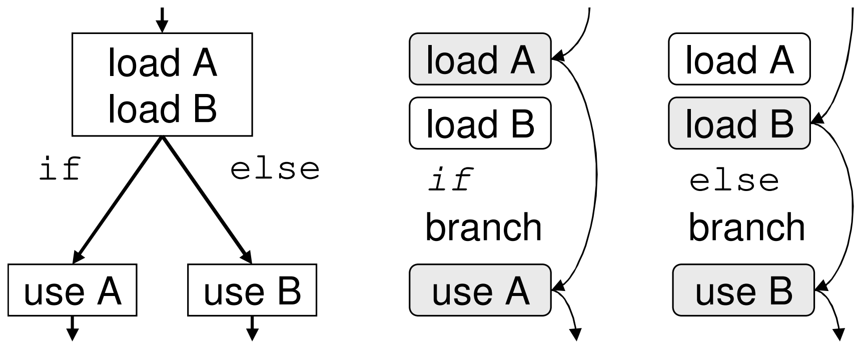
[1]To a limited extent, the compiler can convey some hints about criticality without ISA changes, via its choice of instruction ordering. At least some dynamically scheduled processors, including the Alpha 21264, select an older instruction, i.e. one which appears earlier in the program text, over a younger instruction, when choosing between two ready instructions to schedule on a certain cycle. The QOLD heuristic critical-path predictor, which uses instruction order in identifying critical instructions, could benefit from the instruction ordering produced by a compiler which tries to emit instructions on the dataflow critical path as soon as possible.



(a) Critical path when load hits

(b) Critical path when load misses

Figure III.1: Illustration of variation in critical path due to variable instruction latency.



(a) Control flow graph

(b) Critical path when `else` branch taken.

(c) Critical path when `if` branch taken.

Figure III.2: Illustration of variation in critical path due to variable control flow.

the loads. There are a number of reasons why the static analysis might incorrectly estimate a load's latency. Microarchitectural differences within a processor family, such as different cache sizes, can make a profile collected on one system inaccurate on another.

Figure III.2 illustrates the effects of variable control flow on the critical path. Figure III.2(a) shows a section of a control flow graph. Two loads are performed in the block prior to the branch, but only one load's result is used in each subsequent block. Figures III.2(c) and III.2(b) show dynamic instruction traces. Suppose that the use of A or B is critical in both cases, as indicated by the darkened boxes. Depending on which direction of the branch is taken, either `load A` is critical, and `load B` is unused, or *vice versa.* A static analysis which was limited to a single basic block, or which was based on inaccurate profiles, would incorrectly classify the loads. Together, these examples illustrate how the set of static instructions which are on the critical path can change during the course of a program. Program profiles can be used to identify frequently taken branches, and frequently missing loads. But, the standard list of complaints about profiles applies: variation between training and actual inputs, variation across the course of the program, and reluctance of programmers to create profiles. A dynamic, hardware-based analysis is capable of adapting to changes in load miss behavior and in control flow behavior which a profile cannot capture. A dynamic critical path predictor may implicitly make branch predictions and load hit predictions as a part of making an explicit critical path prediction, thus allowing it to adapt to changes in branch and load behavior, regardless of the reason for the change.

A static analysis has one more drawback. As critical path predictions are applied, the critical path changes, which reveals secondary critical paths. We would like to detect and optimize these too. Exactly which paths are exposed may depend on the details of how the processor uses the predictions. A dynamic predictor has a better opportunity than a static predictor to adapt to the changes caused by its own predictions.

III.C Offline Critical-Path Analysis of Programs

An offline analysis operates on lengthy dynamic instruction traces, either from a simulator, or from program traces collected via special hardware support[33]. We distinguish between an offline analysis, which uses detailed program traces, and a static analysis, which may use branch direction and misprediction statistics, but which does not use detailed program traces.

Studies of instruction level parallelism, [16, 66, 121], consider the execution time of programs constrained only by data dependencies, as well as under various hardware constraints. These studies find that many programs could potentially execute more than 2 instructions each cycle, and in some cases, much more. Typically, a program with a higher level of parallelism, measured in *instructions per cycle* (IPC), will have a smaller fraction of instructions on the critical path. However, these studies do not indicate which instructions are on the critical path.

Perhaps the first study to explore this in detail was one by Tullsen and Calder, [114]. They performed an offline analysis of the critical path by instrumenting executables with a runtime profiler. The runtime profiler maintains a graph of dependencies between instructions. It tracks two sets of instructions: I, the instructions which are dispatched but unexecuted, and T, those recently executed instructions which are reachable from I. At each cycle, their profiler attempts to identify an instruction t in T which completed execution before any other instruction in T began execution. The profiler then records instruction t as part of the critical path, and remove t from T. In addition to data dependence edges, they include dependence edges for branch mispredictions and full instruction window events in the simulator. Their work predates Fields et al., [34], who also models those two dependences as part of a more sophisticated graphical model. Tullsen and Calder summarized this per-dynamic instruction information to yield a single criticality prediction for each static instruction. They then used this prediction to control value prediction, and to guide instruction scheduling. We compare our dynamic techniques to their static predictions in Section V.A.

Srinivasan et al. studied the latency-tolerance of loads in [106]. In their work, latency tolerance refers to the longest latency that a load instruction could have before impacting performance. They find, for many loads, that the latency tolerance of a load does not match the level of the memory hierarchy where its data resides. They analyzed load-latency tolerance using a special simulator which measures the change in behavior when the latency of a dynamic load instruction is increased.

Fields et al. proposed a graph-based model of the critical-path of a program trace which takes into account some processor constraints. This model is described in Section III.E. They used this model to perform an offline analysis of programs to determine which instructions were critical. They also developed a dynamic predictor based on this model, which is described in Section IV.D.1. In [32], they use their graph to study the distribution of slack present in programs.

We built upon their initial offline analysis to measure how much potential for performance improvement lies behind each critical instruction, which we call *tautness*. We refined their model so that the graph remained *robust*: it can be used to determine not only what instructions were critical on a certain microexecution, but also what the program's runtime would be if some latencies are changed. This analysis is described in Chapter VI. In [33], Fields et al extend our extension. They apply our robust analysis to evaluating microprocessor design alternatives.

However, because an offline analysis is, by definition, performed after a program is run, it cannot be practically used to guide optimizations on that program. The offline analyses are also slow. It could be used to guide optimization of future runs of that program, but then it would simply be a static analysis, with all the shortcomings just described in the previous section. Instead, offline analysis is useful as a research tool, for program and processor understanding.

III.D Dynamic Critical-Path Analysis of Programs

In summary, static and offline analyses have several shortcomings. Static analyses may suffer from inacurate profiles, and may be incovenient to perform A dynamic predictor can change its predictions over time, and requires no ISA change, and is applicable to existing software. Therefore, we propose an online, hardware approach to identifying critical instructions. The challenge is to classify instructions with reasonable accuracy, with a modest amount of hardware. The next chapter discusses our approach, and the approaches of others, to dynamic critical-path prediction.

III.E The Critical Path Model of Fields et al.

Fields et al. propose a model for the critical-path of instruction traces which includes not just the data dependencies of instructions, but also dependencies corresponding to control and some resource constraints, such as a finite instruction window. They represent each instruction by several nodes, corresponding to different events as the instruction moves through the pipeline.

In their original model, each instruction is represented by three nodes. Each of these nodes represents that instruction reaching a particular pipeline stage in the machine. These three nodes represent the time when the corresponding instruction is dispatched (*d-nodes*), executed (*e-nodes*), and committed (*c-nodes*). Edges connect events which must occur in order, and the weights on those edges represent latencies between those nodes.

Figure III.3 provides an example of an instruction trace, shown in Figure III.3(a), and its corresponding graphical model, in Figure III.3(b). Each node is labeled with a letter representing its type and number indicating which instruction in the trace it represents.

Edges of type $D_i \rightarrow D_{i+1}$ represent in-order fetching of instructions. In this example, the processor fetches 1 instruction per cycle, so each edge has weight 1. The weight would be 0 between instructions fetched on the same cycle. If an instruction cache miss occurs as a result of fetching a right-path instruction, then the weight is the instruction cache miss penalty.

Edges of type $D_i \rightarrow E_i$ edges represent the delay between dispatching an instruction and executing it. Edges of the type $E_i \rightarrow D_{i+1}$ exist between a mispredicted branch and the next instruction. (Only right-path instructions are included in the graph.) For this example, there are 4 cycles between detecting a branch misprediction and dispatching the next correct path instruction.

Edges of type $E_i \rightarrow E_j$ connect data dependent instructions. The weight is the execution latency of the producing instruction. Edges also exist between a store and a subsequent load to the same address.

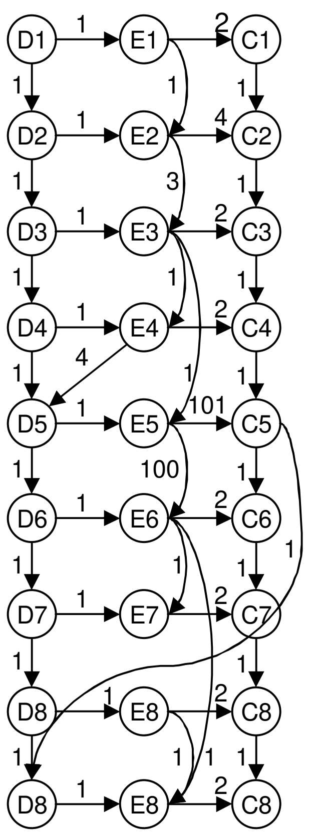
Edges of type $E_i \rightarrow C_i$ are always present, with a weight equal to the execution latency of the instruction plus the number of cycles for a instruction to go from the execution stage to being eligible to commit.

Edges of the type $C_i \rightarrow C_{i+1}$ represent in-order commit. For the processor in this example, only one instruction can be committed per cycle.

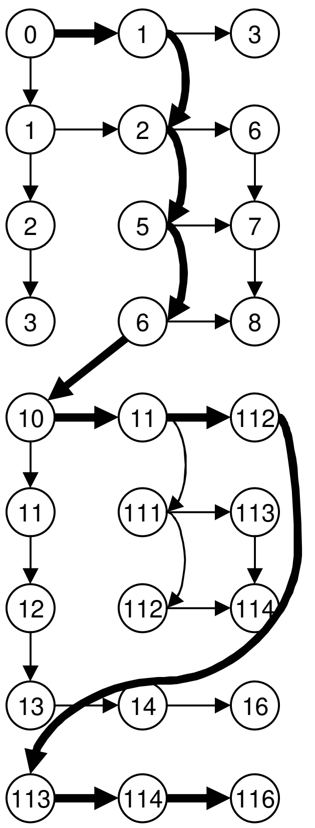
Edges of the type $C_i \rightarrow D_{i+R}$ connect an instruction the oldest instruction in the processor at the time of a full instruction window stall, i, and the next

```
1    immed → R1
2:  LD 8[R1] → R1 (hit)
3:  ADD R1, 8 → R1
4:  BNE R1 (mispredicted)
5:  LD 8[R1] → R1 (miss)
6:  ADD R1, 8 → R1
7:  BNE R1 (predicted)
8:  immed → R2
9:  STO 0[R2], R1
```

(a) Instruction trace.



(b) Nodes with names and edges with latencies.



(c) Nodes with scheduled time, last-arriving edges, and critical edges highlighted.

Figure III.3: Graph of the dependencies of a section of code, following the graphical model of Fields et al. in [34].

instruction to be fetched after the stall, $i + R$. For this example, $R = 4$.

Figure III.3(c) shows a modified graph. Each node is labeled with the longest path length to it from node D_1. If this model accounted for all the limitations of a microprocessor, then the value in each node would also be the cycle at which that instruction would be dispatched, executed, or committed, correspondingly. This time is relative to the cycle when D_1 is dispatched. Only edges which are on the longest path to a node are shown. These are called *last-arriving edges*. The path from D_1 to C_9 using last-arriving edges is the critical path, and the instructions on this path are critical, according to this model. This path is shown with heavier lines.

The model is described in greater detail in [34].

IV

Dynamic Critical Path Prediction

In this chapter we show that it is possible to predict, in hardware, which dynamic instructions are critical. These predictions can be used in many ways. In Chapter V, we demonstrate several ways to use these predictions. For a number of reasons, however, we must necessarily make a prediction, rather than a precise determination of whether an instruction is critical.

Ideally, we would like to be able to model a computer program and the processor it executes on as a graph, and then find the longest path through that graph, and declare that the instructions on that path are critical. For a variety of reasons, this is not possible. First, a graph may not be able to model all the constraints in a microprocessor precisely, although it may serve as a usable approximation. (We address this issue in more detail in Chapter VI.) Nevertheless, there are still some instructions which, when hastened or delayed, affect the runtime of the program; and there are those which do not. We call these critical and non-critical instructions, respectively.

Second, even if we had a precise graphical model of microprocessor behavior, we would need to see the entire program graph, or at least large parts of it, in order to compute the longest path through it. This requirement hardly seems amenable to a low-cost hardware solution. Therefore, when determining whether an instruction is critical, we may have to settle for an approximation which looks at localized effects, or only at small portions of the dependence graph.

Third, even a method which identifies critical instructions by looking at portion of the program trace will still need to wait until the instruction executes to determine its criticality. But, we need criticality predictions early in the pipeline, because they control the instructions passage through it. Fortunately, static instruction tend to be biased: if a recent instance of a static instruction was critical, then the next instance is more likely to be critical than if it was not recently critical. We exploit this to predict whether instructions will be critical, even as they are being fetched.

Thus, the steps in a practical system for dynamic critical path prediction are i) *identifying* which instructions were on the critical path, after they passed through the pipeline, ii) *recording* this information in a table, and iii) *predicting* whether whether instructions are critical or non-critical, based on recorded behavior, as they enter the pipeline. These activities occur continuosly in the processor. All critical path prediction schemes have these 3 aspects, and all schemes record and predict in the same way. They differ in how they identify critical instructions.

Section IV.A describes the common aspects of critical path prediction. Section IV.B describes our initial approach, which was part of the first proposal for critical path prediction. Section IV.D describes critical path prediction schemes, and related techniques, which were proposed by other research groups. Section IV.C describes a pair of improved critical path predictors. Section IV.E compares our designs, and a design by another group of researchers. In this chapter, we are concerned with designing and evaluating critical path predictors. However, a critical path predictor is not useful by itself. It is used to guide some other hardware policy or optimization. Practical applications of critical path

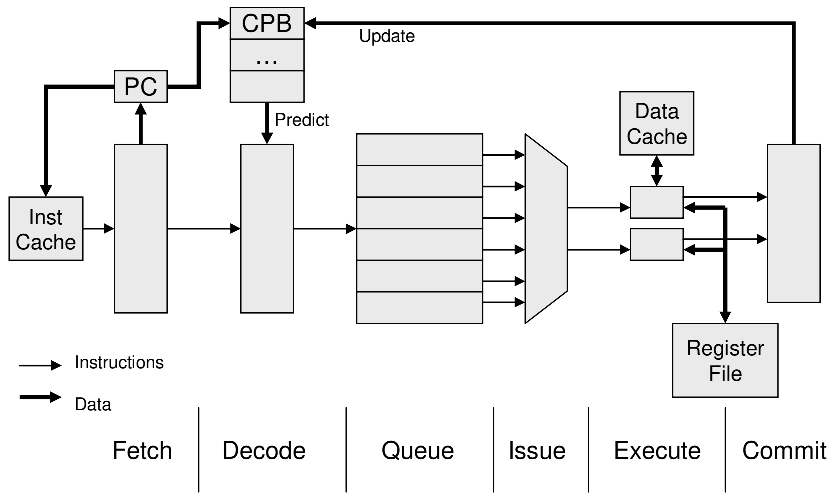


Figure IV.1: A pipeline with Critical Path Buffer.

predictions are discussed in Chapter V.

IV.A Common Aspects of Critical Path Predictors

Our critical path predictor designs and those of other researchers use the same mechanism to record and identify critical instructions: a table of counters, which we call a *Critical Path Prediction Buffer* (CPB). Figure IV.1 shows a processor pipeline with a CPB.

Critical path prediction, like branch prediction and value prediction techniques, is based primarily on the previous history of an instruction. A PC-indexed table of saturating counters is updated according to an instruction's prior trips through the processor, and is queried when the instruction is next fetched.

All the critical path prediction schemes follow the same process. Some mechanism is used to *identify* critical instructions. For our prediction schemes, an instruction meets a *criterion* as it passes through the pipeline, and that instruction is *marked* to indicating that it may have been critical. When and if an instruction commits, the CPB is updated based on whether the instruction is marked. A saturating counter corresponding to that instruction is incremented if the instruction was marked, or decremented if it was not marked. In the token-passing scheme of Fields et al., a profile is performed using special hardware. The result of a profile, which spans several hundred instructions, is used to update the CPB entry for a single instruction. These mechanisms differ in the rate at which the buffer is updated.

When the instruction is fetched, it is *predicted* to be critical if the counter in the CPB is above a threshold value. Otherwise it is predicted to be non-critical. An additional bit in the decoded instruction would indicate its criticality.

Except where noted otherwise the counters in the CPB have the following format. Each counter is 6 bits. The counters are incremented by 8 during commit when an instruction is identified as critical, and decremented by 1 when it is not. Instructions are predicted as critical when they have a counter value above 8. The counters are *saturating*: they cannot be decremented past 0 or incremented past 63. This asymetrical increment/decrement scheme provides several benefits, which are discussed more later. It causes the predictor to continue to predict an instruction as critical, even when optimizing makes it seem non-critical, and is biases the predictions toward predicting as critical those instructions which vary in their criticality.

Our preliminary investigation of settings for increment, decrement and threshold, found the above values to perform well over the critical path prediction heuristics we examined. We consider the effect of other increment and decrement sizes in Section IV.E.4. The counters are indexed by instruction PC, and are not tagged, to save space.

We use the term *critical path predictor* to refer to the combination of a mechanism to identify critical instructions, and the CPB mechanism. To clarify: the term *predicted as critical* and the term *marked/identified as critical* mean different things.

Again, our critical path predictor includes a table of saturating counters. A table of counters has long been used to predict branch outcomes [101]. Likewise, a load hit-miss predictor [129] predicts whether individual load instructions will hit or miss in cache. While load misses are often critical, they need not be. Lipasti and Shen, [70], proposed predicting the dependences between instructions for the purpose of pipelining dispatch, but the relationship of the predictions to the critical path was not considered.

IV.B Heuristic Critical Path Predictors

This section describes *heuristic* critical path predictors. We consider several heuristics for identifying critical instructions as they pass through the pipeline. A heuristic critical path predictor relies on the behavior of individual instructions as they pass through the pipeline to identify which instructions are critical. We term them heuristics because they are certainly subject to error. However, most of the heuristics we consider have simple hardware implementations, and provide a useful classification of instructions.

This section is organized as follows. We introduce the heuristic approach in IV.B.1 with a case study. Our experimental methodology for this Section is described in IV.B.2. We describe the heuristics in detail in IV.B.3, and evaluate their effectiveness in IV.B.4.

IV.B.1 Identifying Critical Instructions

We use a simple code example to demonstrate the importance of finding the critical path, and to give insight into how simple heuristics might be used to recognize those critical path instructions. Figure IV.1 shows the compiler-generated code for a simplified (for clarity) version of Livermore Loop 23, which

	Code		**SC IPC**	**IQ latency**	**Cycles Oldest in IQ**
	ldt	f1, 8000(t3)	1.02	1	0
	ldt	f10, 0(t1)	1.02	1	0
	ldt	f11, 8(t3)	1.02	1	0
	ldt	f12, 0(t4)	1.02	1	0
	addq	t2,0x1, t2	1.02	1	0
	cmplt	t2,a1, t7	1.02	2	0
	lda	t1, 8(t1)	1.02	1	0
	lda	t4, 8(t4)	1.02	1	0
	lda	t5, 8(t5)	1.02	1	0
	lda	t3, 8(t3)	1.02	1	0
	mult	f1,f10, f1	1.02	3	0
B	**ldt**	**f10, -16(t3)**	**5.51**	**286**	**1**
	mult	f11,f12, f11	1.02	4	0
	ldt	f12, -8(t5)	1.02	1	0
	addt	f1,f11, f1	1.02	8	0
	mult	**f10,f12, f10**	**5.27**	**288**	**2**
	ldt	f12, -8(t3)	1.02	1	0
	addt	**f1,f10, f1**	**6.74**	**290**	**4**
	subt	**f1,f12, f1**	**6.06**	**294**	**4**
	mult	**f1,f0, f1**	**5.66**	**298**	**4**
	addt	**f12,f1, f1**	**4.80**	**298**	**4**
A	**stt**	**f1, -8(t3)**	**-**	**302**	**4**
	bne	t7, ...	-	1	0

Table IV.1: Assembly code for a simplified version of Livermore Loop 23. SC IPC is the throughput achieved by the loop if the associated instruction is removed from the dependence chain (by not making its dependents wait for its result). The following columns are the average number of dependent instructions in the processor when the instruction is issued, the average stay in the instruction queue, and the average stay at the bottom of the IQ.

has one loop carried dependence (besides the induction variables). This dependence is through memory, from instruction A to instruction B in the next iteration. The instructions along the data-flow path from the load to the store form the critical path for this code, and are shown in bold.

For each instruction, we found the short-circuit IPC (labeled SC IPC in the table). Short-circuit IPC with respect to a static instruction s is the throughput that this code achieves if all instructions that depend on s do not have to wait for s. The short-circuit treatment is meant to represent the most drastic performance-improving action, or *optimization*, that might be applied to a single instruction, by an unspecified hardware performance enhancing mechanism. The SC IPC is shown for each of the 20 instructions which produces a value. The IPC with no changes is 1.02. So, for 15 of those 20 instructions, applying an idealized optimization has no effect. For the remaining 5, the optimization results in a large increase in throughput. This set of instructions constitutes the critical path, and just as importantly, the remaining instructions are non-critical. An optimization applied to a non-critical instruction is wasted.

This code segment, highlight several important points. The longest data flow path through the static instructions is different from the critical path. A compiler analysis of the critical path, as discussed in Section III.B, which does not consider dependencies through memory and across iterations, would not find the critical path. Also, note that instruction type (load vs. arithmetic) provides no clues as to the critical path in this example.

In studying heuristic predictors, choose not to attempt to explicitly track all dependence chains and identify the ones that matter. Instead, as this example shows, the behavior of an instruction as it moves through the pipeline can indicate where the critical path is. For example, critical-path instructions, and their dependents, tend to get stalled in the instruction queue, and often become the oldest instruction in the queue at some point. The columns labeled “IQ latency” and “Oldest in IQ” in Figure IV.1 show for each instruction, the average number of cycles spent in the instruction queue, and at the bottom of the instruction queue, respectively. These numbers correlate well with the critical path.

It is easier to track a heuristic like this in hardware than a detailed chain of dependencies. And, the criticality of instructions is determined not only by data dependencies, but also on specifics of the processor. For example, if the instruction window of the processor is too small to hold an iteration of this example loop, the critical path through the loop may change. The data-dependencies do not change to reflect this, but the behavior of individual instructions can.

Several things make finding the critical path more difficult in the general

Benchmark	Input	Fast Forward
lisp	ref	1000000000
compress	`bigtest.in`	1000000000
go	`5stone21`	1000000000
perl	`scrabbl`	1000000000
ijpeg	ref	100000000
gcc	`1stmt.i`	0
burg	`rrh-mot`	0
delta-blue	long	0
mpegplay	`sukhoi.mpg`	100000000

Table IV.2: Benchmarks used in Sections IV.B, V.A, and V.B.

case than in this example: irregular control flow, large instruction working sets, branch misprediction, and variable instruction latencies. Despite that, we find that heuristic critical-path predictors still provide useful predictions for a range of programs.

IV.B.2 Methodology

Table IV.2 summarizes the benchmarks used in this section and in Sections V.A and V.B. The first 6 benchmarks come from the SPEC 95 integer suite, and their inputs come from the reference set. These benchmarks are compiled with the DEC CC compiler at –O4. *Mpegplay* is an IBS benchmark [119]. *Burg* is a C++ parser generator. *Delta-blue* is a C++ constraint solution system. Both Burg and Delta-blue have significantly higher data cache miss rates than the other benchmarks. The benchmarks were fast-forwarded the number of instructions indicated in Table IV.2 before being simulated for 300 million instructions.

Execution is simulated on an out-of-order superscalar processor model which runs unaltered Alpha executables. The simulator is based on the `smtsim` simulator [113], but enhanced to include a critical path predictor, and to take advantage of various critical path-aware optimizations. The simulator models all reasonable sources of latency, including caches, branch mispredictions, TLB misses, and various resource conflicts, including renaming registers, queue entries,

Parameter	Value
Fetch width	16 instructions per cycle
Branch predictor	Same as Alpha 21264
Branch Target Buffer	256 entry, 4-way associative
Active List Entries	1024
Functional Units	12 Integer (8 also load/store), 6 FP
Instruction Queues	128-entry Int, 128-entry FP
Registers	200 Int, 200 FP
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache	4 MB, 2-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 98 cycles (if no contention)
Instruction Latencies	Based on Alpha 21164

Table IV.3: Processor parameters used in Sections IV.B, V.A, and V.B.

etc.

The simulated processor configuration shown in Table IV.3 was used for the studies in Sections IV.B.4, V.A and V.B. The configuration models a future wide superscalar out-of-order machine, with an aggressive fetch unit, a large instruction window, and a large unified renaming unit. The L1 caches modeled are more modest, to compensate for the relatively small memory footprint of most of our benchmarks. The fetch unit can fetch up to 16 instructions per cycle from up to three basic blocks per cycle. This simulates the behavior of an effective trace cache [89].

The processor model used in our simulator has 9 stages. During the fetch stage, instructions and predictions which were requested in the previous cycle arrive. After decoding and register renaming, integer and floating-point instructions enter separate instruction queues. The instructions reside in the queues in-order. Every cycle, the oldest instructions which have their dependences satisfied are issued (out-of-order), until no more instructions are ready or no more functional units are available. They require one stage to read register values before they can begin execution. After execution, they go through one stage to write back registers. The instructions remain in the commit stage until they can be committed in order. This pipeline is similar in basic structure to the

Alpha 21264, described in [24].

IV.B.3 Heuristics

In understanding this new architectural technique, we want to separate the effectiveness of the technique from any aliasing effects that might occur in a small prediction table. Therefore, we initially evaluate the predictors in this section with a relatively large 64k-entry direct-mapped of counters. We consider the effect of a smaller table in Section IV.E.4.

Critical Path Marking Techniques

In this chapter, we propose five different criteria that might be used to mark each instruction as either on the critical path or not on the critical path. We evaluate each criterion individually; only a *single* CP criterion is applied during a particular simulation. Some are trivial to implement, others might be more complex. Initially, we are more interested in what works than the complexity of the implementation. Our approach was to evaluate many criteria which we thought might indicate the criticality of instructions. The criteria are summarized in Table IV.4. What follows is a more detailed description of each criterion and the rationale behind it. This is actually a subset of the predictors we investigated, but includes those that were interesting either because of their performance or the intuitiveness of the approach.

The QOld criterion is based on the observation that instructions on the critical dependence path will typically reach the bottom of the instruction queue before they issue. Likewise instructions which are ready to execute as soon as they enter an occupied instruction queue are likely to be non-critical. Any instruction which reaches the bottom of the queue becomes the oldest instruction. This instruction has dependences that exceed (in time) the dependences of all prior

Criterion	Description
QOld	"**OLD**est instruction in **Q**ueue" Each cycle, the oldest instruction in an instruction queue is marked, if it is not ready to issue.
QOldDep	"**DEP**endence with **OLD**est instruction in **Q**ueue" Every cycle, each instruction which produces a value consumed by the oldest instruction in the queue is marked if it is still active.
ALOld	"**OLD**est in **A**ctive **L**ist" Each cycle, the oldest instruction in the active list (re-order buffer) is marked.
QCons	"Most **CONS**umers in **Q**ueue" Each cycle, the instruction is marked whose result is used by the most instructions in the instruction queue.
Freed3	"**FREED** up at least **3** instructions in queue" If the completion of execution of an instruction makes at least three instructions in the instruction queue ready to execute, then the completing instruction is marked.

Table IV.4: The criteria used in this study to mark instructions as critical path, and a brief description of each.

instructions in the instruction stream (for that queue, integer or floating point) [1]

Whereas QOld marks the oldest instruction in an instruction queue, the QOldDep criterion marks the one or two instructions upon which it is dependent. In other words, if the instruction at the head of an instruction queue has source registers x and y, then we will try to mark the instructions which produce x and y. However, if x has already left the pipeline we do not mark it, since x's entry in the CPB would have already been updated when x committed. Therefore, QOldDep marks zero, one, or two instructions per cycle. This criterion attempts to mark the instructions that are currently causing instructions to back up in the instruction queues. This is one step earlier in the critical path dependence chain than the oldest instruction in the queue (QOLD).

The ALOld criterion is based on the observation that the oldest active instruction in the machine is likely to be one that was stalled for some reason, either because of dependences or because it took a long time to execute. The

[1]The instruction queue is same as the instruction scheduling window mechanism in the MIPS R10000 or Alpha 21264. There are actually two separate instruction queues in the simulated architecture: one for integer operations (including loads and stores) and one for floating-point operations. For simplicity, we will speak as if there were only one instruction queue, but in fact, the criteria which involve instruction queues are applied independently to each queue.

active list has an entry for every instruction in the pipeline, waiting to commit in order. The oldest instruction in the active list is usually one that completed execution later than all prior instructions.

The QCons criterion marks the one instruction, among those completing execution, which has the most direct consumers in the instruction queue. We define a consumer as an instruction that will read the value written by this instruction. In the case of a tie, the earliest instruction in the instruction stream is marked. The QCons criterion is based on the observation that instructions that have a large dependence fan out are more likely to be on the critical path. Bahar, et. al. [37] tried measuring processor performance over very short time scales to allow the identification of non-critical loads, but found that counting the number of consumers of a load was a better metric. This corresponds to the QCons criterion.

The Freed3 criterion is similar to the QCons criterion, but it only counts consumers which become ready to execute immediately (they are *freed* by the executing instruction). This criterion is implemented as a threshold mechanism. It marks all instructions which wake up, or "free up", 3 or more instructions in the instruction queue. The idea of scheduling instructions earlier which have a high fan-out has been applied to static instruction scheduling in compilers [41].

An instruction that stalls in the instruction queue or has a large execution latency is likely to accumulate more instructions in the queue waiting for its completion. Therefore, QCons and Freed3 indirectly account for the time that an instruction spends in the IQ. Freed3 and QCons will obviously miss some instructions on the critical path that have only a single output dependence.

IV.B.4 Results

Evaluating critical path prediction is more difficult than evaluating other prediction techniques. This difficulty stems from two significant differences between CP prediction and other predictors. First, in CP prediction it is more difficult to verify the accuracy of a prediction. Second, when CP predictions are used to direct optimizations, these optimizations will affect future CP predictions.

There are two steps in a branch predictor: prediction and verification. The true outcome of the branch is used to verify the prediction and to train the predictor. In critical path prediction, however, we can only verify whether the instruction again satisfied the criterion; we cannot verify whether or not the instruction was actually on the critical path. The predictor is only predicting that the criterion will be met again in the future. Therefore, for critical path prediction to work, we must meet two conditions. First, the predictor must accurately predict which instructions will meet the marking criterion. Second, the marking criterion must be a good heuristic method for identifying critical path instructions. In evaluating our techniques, we measure two different aspects of CP prediction. In section IV.B.4, we asses the *predictor accuracy*; how accurately does the predictor predict whether instructions will meet the marking criterion. In section IV.B.4 we measure the *criterion effectiveness*; how well do the predictions indicate which instructions are in fact critical.

The second difficulty may be referred to as the *feedback* problem. Namely, prior predictions affect future predictions. In bimodal branch prediction, the prediction used for the branch will not affect the update of the counter. In critical path prediction, an instruction that is predicted as critical will be optimized (e.g., value predicted, sent to a different cluster, etc.). After being optimized, it may no longer be on the critical path, and it may not be marked as critical. However, if it is subsequently not optimized, it may again appear on the critical path. This

Criterion	Percent of Instructions Marked	Percent of Instructions Predicted	Percent Non-CP Prediction Accuracy	Percent Positive Prediction Accuracy
QOld	14	26	99	49
QOldDep	17	33	99	50
ALOld	15	35	99	36
QCons	6	16	99	36
Freed3	5	7	99	64

Table IV.5: The percent of executed instructions that each technique marks and causes to be predicted, as well as the accuracy with which each predictor predicted the same behavior used to mark instructions.

effect is discussed more in section IV.B.4.

Measuring Prediction Accuracy

This section examines the degree of self-correlation (or repeatability) of the prediction criteria — that is, if event A is used to mark critical instructions and update the predictor, is the corresponding predictor actually a good predictor of event A? If not, it is unlikely to be a useful criterion.

To measure this self-predictability, the simulator was set only to mark and predict instructions; no actions were taken based on the predictions. What was measured is how often an instruction, which was predicted to be on the critical path, was again marked as a critical path instruction.

Table IV.5 shows the results for each CP algorithm, averaged over all benchmarks. The first column lists the names of the criteria tested, as described in section IV.B.3. The column labeled "Percent Instr. Marked" shows the percentage of dynamic instructions that had their CP marked bit set. The column labeled "Percent Instr. Predicted" shows how often any dynamic instruction had its CP predicted bit set. Remember that an instruction has its predicted bit set if its counter, in the PC-indexed Critical Path Buffer, is above 8. The column marked "Percent Non-CP Prediction Accuracy" measures what fraction of dynamic instructions that are predicted as "not on critical path" do *not* trigger

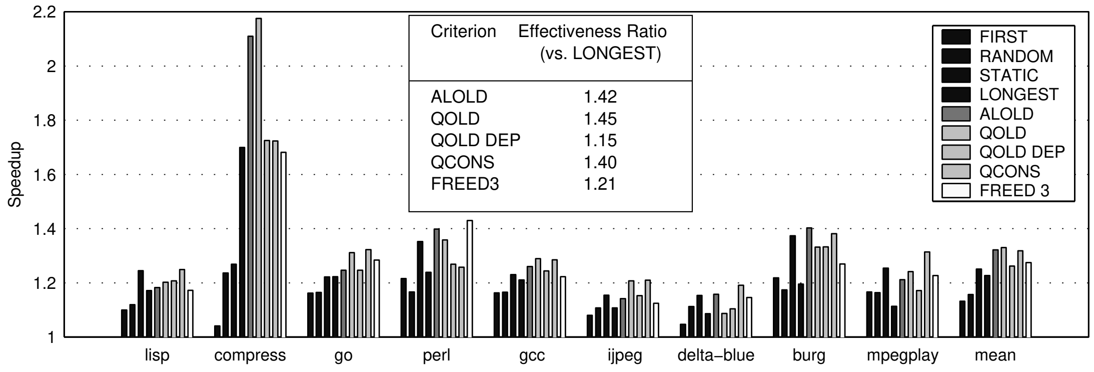


Figure IV.2: The performance resulting from breaking the dependences of critical path instructions.

the marking criterion again. The column marked "Percent Positive Prediction Accuracy" measures what fraction of dynamic instructions that are predicted as being on the critical path have their CP marked flag set again the next time they are executed.

The results demonstrate that our predictors are intentionally liberal. One reason for this is to identify instructions only occasionally on the critical path. For example, on a load with a 20% miss rate that is only on the critical path when it misses, we might do best to always predict it on the critical path. This assumes that the cost of a wrong positive prediction is typically less than the cost of not predicting the instruction as being critical. Note that for the 65-93% of instructions predicted as not being on the critical path, the predictors are virtually always right.

Measuring Prediction Effectiveness

This section evaluates the effectiveness[2] of our marking criteria in indicating which instructions are on the critical path. One approach would be to compute the critical path of a program by finding the longest chain of dependent

[2]The effectiveness ratio shown in the figure is defined in V.A.

instructions in a trace of the program, and to compare these instructions with those that are predicted by the CP predictor. There are several down-falls to this approach:

- The statically-determined critical path depends not just on dependences, but also on the idiosyncrasies of the processor, including queue sizes, active list size, number of renaming registers, and even on the input used when running the program.
- When the critical path information is used to optimize certain instructions, the optimizations can change the critical path, and the critical path predictor needs to adapt to the changes in the critical path caused by its previous predictions. The statically-determined critical path does not account for these changes.

To evaluate performance we will again use the approach from section IV.B.1, which focuses on the actual performance when the critical path prediction is used to change execution. In this section, we apply an ideal, generic optimization to compare several proposed predictors outside of the context of a specific optimization; the next section applies more realistic optimizations.

In this experiment, each cycle in which instructions are fetched, one instruction from the fetched block is chosen to execute with no output dependence stalls. That is, subsequent instructions that depend on this instruction will not have to wait for this instruction to execute. This emulates optimizations that break data dependence chains, such as value prediction and instruction reuse, but without presupposing exactly what optimization it is or which instructions it would work on. The choice of which instruction to select is based on the critical path prediction.

Figure IV.2 shows the speedup achieved on this test for the various dynamic predictors. The speedup is relative to the execution time with no optimization. We also provide the following measurements for comparison:

- FIRST: Always select the first instruction fetched this cycle.
- RANDOM: Pick an instruction randomly each cycle from the instructions fetched.
- STATIC: We pre-compute the critical path of the program by identifying the instructions which are on the longest chain of dependences in the program using profiling [114]. The profiler computes a dynamic critical path, accounting for cache and branch effects as well as a limited instruction window size. While a single, complete dynamic path is identified, the tool creates a static summary of each instruction's contribution to the dynamic critical path. The most critical static instructions, accounting for 98% of the dynamic path, are then statically identified as critical for the purposes of the STATIC predictor in these simulations. Each cycle, then, a statically marked instruction is chosen from the fetch block to be optimized, if possible.
- LONGEST: The instruction with the longest estimated execution latency is chosen. The latency is "estimated" because the latency of loads varies. The hierarchy of latencies we assume is based on Alpha 21264 latencies. We use a static estimate for load latency which places it lower than integer multiply and most floating point arithmetic operations, but above all other integer operations. For the integer-intensive applications shown here, then, LONGEST often amounts to "choose the first load." Exceptions are `mpegplay` and `ijpeg` which have a fair number of integer multiply and floating-point instructions. We also tested a different version of LONGEST which prioritized loads over integer multiply and floating point instructions, but it did not perform as well.

We see that in almost all cases, the use of critical path prediction consistently results in greater speedup than the non-dynamic FIRST and RANDOM mechanisms. We found that on every benchmark, ALOLD, QOLD, QOLDDEP, and QCONS performed better than LONGEST. FREED3 was slightly worse on lisp and compress, but better then LONGEST on the other seven benchmarks. Additionally, on each benchmark, at least one of our dynamic predictors performed better than STATIC. This confirms that our dynamic predictors are adapting to changes in the critical path (chiefly caused by the optimizations themselves) in ways that the STATIC predictor cannot. Note that the benchmark and input files used to generate the static profile are identical to those used in the simulations. In a practical use of static profiling, differences between the inputs used for generating the static profile, and for actual execution would likely reduce the performance of the STATIC method. In cases where the dynamic predictor, or the STATIC predictor, don't select a single critical instruction with a fetch block, the LONGEST mechanism is used as a tie-breaker [3].

Counter Format and Prediction Persistence

In some cases, predicting an instruction as critical path (and applying some optimization) causes that instruction to no longer be on the critical path. However, this does not mean that we should no longer consider the instruction as critical. We'll refer to the predictor's natural inclination to start decrementing an instruction's CPB counter as *forgetting* a prediction. We can minimize the CPB's tendency to forget by incrementing the CPB counters by a large amount when an instruction is on the CP and decrementing by a small amount when

[3]The dynamic predictors and the STATIC predictor may predict multiple critical instructions within one fetch block. In this case, a tie-breaker is needed to select one instruction from within the block. We use estimated latency, as described for the LONGEST mechanism, to break ties in these cases. Similarly, the STATIC or the various dynamic mechanisms may predict that no instruction is critical. For this evaluation, where there is no penalty for misprediction, it makes sense to always make some value prediction, since the critical path predictor could be wrong. Again, in this situation, a tie-breaker is used.

not. In the previous experiments, we increment by eight and decrement by one, partially to avoid forgetting. Any instruction with a counter greater then eight has its predicted bit set. In the worst case, a CP instruction gets retried every eighth execution to confirm its criticality.

Not all of the marking criteria are affected in the same way. In particular, QCons and Freed3 always forget because a successful optimization eliminates the dependences. On the other hand, when an instruction's result is, for example, value-predicted, that instruction must still execute to verify the prediction. Consequently, we would expect that ALOld and QOld would be less prone to forgetting. To verify this, we reran the dependence-breaking experiment of the previous section but with a a more forgetful counter, incrementing by two and decrementing by one. In these experiments, QOld and ALOld both performed better with the more forgetful counter, but the others (QOldDep, QCons, and Freed3) all performed better with the original increment-by-eight, confirming that they need the help of the confidence counters to force prediction persistence.

IV.B.5 Analysis

Of the heuristic predictors, we have found the QOld and ALOld most effective. The heuristic predictors are simple to implement, and work well enough to demonstrate the potential of critical path aware optimization, in Chapter V. However, there is room for improvement.

In [32], Fields et al. found that their token-passing predictor critical path consistently performed better than the ALOld predictor, but that both performed better than policies which do not take criticality into account, when evaluating critical-path aware power reduction.

None of the heuristic critical path predictors specifically detect instructions which lead up to a mispredicted branch as being critical. However, branch mispredictions most likely affect processor performance, and thus any instructions which delay resolution of a mispredicted branch are likely to be critical too. The QOld and ALOld predictors will tend to identify chains of dependent instructions. However, they will favor chains which link an instruction to its oldest dependent. For example, say a load instruction i reaches the head of the queue. The QOld predictor will mark it. Often, the instruction which reaches the head of the queue on the cycle after i is issues is dependent on i. If so, then it will be the oldest dependent of i, since the instructions are assumed to be sorted in program order within the queue. Younger dependents of i can also be marked in some circumstances, but the oldest dependent may be marked disproportionately. The following sections consider alternative approaches to predicting the critical path.

IV.C Iterative Critical Path Predictors

A heuristic predictor's simplicity permits a low-cost, unintrusive, hardware implementation. It is simple because it does not explicitly track dependence chains. However, this simplicity may limit its accuracy, as discussed in the previous section. With this in mind, and with the benefit of experience, we developed a new type of critical path predictor: the *iterative* critical path predictor.

IV.C.1 Approach

The design of the iterative predictor is based on several observations:

1. If only one instruction is active in the pipeline on a certain cycle, then that instruction is critical.

2. An instruction which wakes up a critical instruction is also critical.

Observation 1 is not sufficient to identify all critical instructions, since an instruction can be critical, even though other instructions are being processed concurrently. However, as soon as one instruction is identified as critical, many more instructions can be identified on the critical path leading up to the first instruction, using observation 2. This process is analogous to an inductive proof. Observation 2 is like an inductive hypothesis, and observation 1 allows us to identify base cases for induction.

We considered two types of events that occur in the processor which satisfy observation 1.

A. If on any cycle, the instruction queue is full, and instruction i is the only instruction executing, then i is critical.

B. If, after a branch misprediction, all pre-branch correct-path instructions older than the branch complete execution before any post-branch instructions begin execution, then the branch instruction is on the critical path.

These conditions are specifically designed to be strict: they may ignore many critical instructions. We require that the instruction queue be full in case A. If it were not, then the processor could be in the process of fetching other instructions. Those other instructions may be critical, and the ones currently executing may not be. In condition B, the processor is only doing one thing: fetching instructions. The only way to get it to fetch those instructions sooner is to detect the mispredicted branch sooner. Condition B is also too strict. Some branch mispredictions may be spanned by a concurrent operation, but the branch misprediction can still delay fetching critical instructions.

Once an instruction is identified as being critical, by condition A or B, its counter in the CPB is incremented, just as described for the heuristic prediction scheme. When a subsequent instance of that instruction is fetched, it will be predicted to be critical, and flagged as such in the processor. Whichever instruction produces the last operand for this predicted-critical instruction will in turn be identified as critical, and its counter incremented in the CPB. The first instruction which is identified by observation 1 is like a seed from which a longer critical path can be grown. Each time this instruction passes through the processor, the path may grow longer by one instruction. In fact, because of the hysteresis present in the counters in the CPB, combined with variability in control-flow, and in which operand is last-arriving for a given instruction, this path need not be simple chain, but can fork as well. Instructions which meet neither the last-arriving or seed criteria have their CPB entry decremented.

The implementation of observation 2 deserves some discussion. A simple two-operand instruction, such as an addition instruction, waits in the instruction queue until both of its operands are ready. Whichever of those operands is ready second is the *last-arriving operand*. The instruction which produced that last-arriving operand is also identified as critical. If both operands were ready before the instruction came into the instruction queue, then neither producer is critical. For load instructions, the last-arriving operand is either the register operand, or the memory operand. If the memory operand is forwarded from a store instruction in the load-store queue, then it may be last arriving. Including load-store bypassing is important to the accuracy of the critical path predictor. Since processors already need to detect this condition, it should not be too hard to implement.

Table IV.6 shows how many instructions meet either condition A or condition B. We call these instructions *seed* instructions, since they provide a seed from which to grow a critical path. The numbers are shown for any instruction which meets condition A, by executing alone, and for branch instructions which met condition B. The numbers represent seed instructions identified per one million instructions executed. Although the solo-executers, instructions which meet

Benchmark	Abbrev.	Solo-executers (per 10^6 inst)	Branch seeds (per 10^6 inst)	Total seeds (per 10^6 inst)
ammp	amm	56699	281	56980
applu	app	148	4	152
art-110	art	95	161	257
crafty	cra	211	4322	4534
eon-rushmeier	eon	701	1947	2649
equake	equ	908	0.03	908
galgel	gal	122	0.44	122
gap	gap	472	3796	4269
gcc-166	gc1	148	2394	2543
gzip-graphic	gzg	559	3482	4041
mcf	mcf	2113	4577	6690
mesa	mes	97	227	324
mgrid	mgr	1728	0.01	1728
parser	par	7008	3162	10171
perlbmk-makerand	pem	1576	8892	10469
swim	swi	869	0.06	869
twolf	two	3080	4989	8069
vortex-2	vo2	704	793	1497
vpr-route	vpr	1820	6181	8002
mean	avg	4161	2379	6541

Table IV.6: The number of seed instructions identified by the iterative predictor for executing alone, or being a branch misprediction, and meeting other criteria.

condition A, are more common overall, the instructions meeting condition B are more frequent in some benchmarks. Again, these conditions are specifically designed to be strict. They are sufficient but not necessary conditions for an instruction to be critical. When combined with observation 2, however, many more critical instructions can be identified.

Some benchmarks have very few total seeds. In many cases, these are scientific applications have highly predictable branches, and thus few branch mispredictions; and they have high-levels of instruction level parallelism and thus there are almost always multiple instructions executing at once. Thus, neither of the seed criteria are met.

For the cases when the iterative predictor cannot find a seed or grow a path, it makes sense to fall-back on a different method of identifying the critical path. Therefore, we also propose a *hybrid* predictor. The hybrid predictor combines the iterative predictor, and a heuristic predictor. We use QOld. When the iterative predictor is working, only the iterative predictor is used. When the iterative predictor does not identify any critical instruction by either method (seed or last-arriving) for 100 committed instructions, then the hybrid predictor falls back into heuristic mode. While in heuristic mode, it does not use the last-arriving rule to mark instructions. It only uses the QOLD criterion. Anytime a seed is found, the hybrid predictor reverts back to iterative predictor mode.

We evaluate the performance of the iterative predictor and the heuristic predictor in Section IV.E.

IV.D Other Critical Path Predictors

We first proposed critical path prediction of instructions and suggested its broader applications in [117]. There are several bodies of work which predate our work, and which relate to identifying the critical path of programs. First, some compilers use critical-path based instruction scheduling algorithms. This is discussed in Section III. Second, Bahar and her colleagues classified data and instruction cache lines as critical or non-critical [7, 37]. Following our initial publication, Fields et al [34] proposed a different critical path predictor, and two groups, Racvik et al. [87] and Srinivasan et al. [105], proposed load-instruction-only criticality predictors.

Bahar et al. [7], classified instruction cache misses as being critical or non-critical, based on the occupancy of the instruction queue at the time of the miss. They observed that instruction cache misses are often not critical when the instruction queue is well occupied. Fisk and Bahar [37] classified data cache lines as critical or non-critical based on the IPC (number of instructions executed per cycle) around the time that the data was accessed. Their work differs from ours in that they classify data or instruction cache lines as being critical, while we classify individual instructions as being critical.

Srinivasan et al. proposed a method for identifying critical loads. They consider a load instruction which misses in cache to be critical if its value feeds into a mispredicted branch, if it feeds into another load which misses in L1 cache, or if the instruction issue rate falls below a certain threshold immediately after the load miss. Racvik et al. call a load instruction “non-vital” when its result is not used immediately by any instruction. In this way, they identify some non-critical load instructions. Both of these techniques are limited because they only predict the criticality of load instructions.

IV.D.1 Token Passing Predictor

Fields et al. presented a graphical model which incorporates both data dependencies and some hardware constraints. Their model represents each dynamic instruction with 3 nodes in a directed graph. We describe their model in Section III.E. They also present a hardware scheme which approximates, which high accuracy, whether a recently executed dynamic instruction was on the critical path of this graph. Their hardware scheme approach relies on several ideas. First, the critical path consists of a path through the graph which only traverses *last-arriving edges*. In the context of applying the longest-path algorithm to a graph [25], the last-arriving edge to a node is the one which last updates the distance to that node. In the context of a processor, a last-arriving edge is the dependency which is satisfied last. Figure III.3(c) shows the last-arriving edges of a graph, and the critical path. Fields uses a *token-passing* technique to determine whether a recently executed instruction is on the critical path. Conceptually, a token is placed in the e-node of the instruction whose criticality is to be measured. Any node t accepts a token from node s if (s, t) is the last-arriving edge to t. A node with a token offers a copy of the token to all its dependents. The token will reach the sink of the graph if and only if there is a path to the sink of the graph which traverses only last-arriving edges. In practice, an instruction is declared critical if it reaches some instruction which is at least several hundred instructions past the starting instruction. In practice, the passing of the token can be simulated with a small memory, controlled by information about last-arriving edges for each instruction. Because their graph includes 3 nodes for each instruction, it is possible to say that fetching an instruction is critical, as opposed to executing the instruction. However, all applications of critical path prediction apply to *execute-critical* instructions, rather than *fetch-critical* instructions. Therefore, we simple use the term "critical" to mean what Fields, et al. call execute-critical. We compare their predictor to ours later in this chapter.

IV.E Comparison of Critical Path Predictors

In this section, we compare the performance of four critical path predictors. We evaluate our iterative predictor, our two best heuristic predictors, and token-passing predictor from Fields, et al., [34].

IV.E.1 Methodology

As mentioned before, precisely defining which dynamic instructions are critical is difficult. However, we find that there are clearly some instructions which benefit from being *optimized*, or made to execute more quickly; and there are clearly those that do not benefit, where benefit is defined as the whole program executing faster. In this section, we take the approach that a critical path predictor is doing well if it picks a small set of instructions, which, when optimized, result in a large reduction in program execution time.

To evaluate the predictors, we modified our simulator to add one cycle of execution latency to all instructions. The performance of each benchmark in this configuration serves as a baseline[4]. Also, we ran simulations using each

[4]We increased the latency of all instruction before decreasing so that single cycle instructions would not become zero cycle instructions.

Name	Input	Fast Forward Instructions ($\times 10^6$)
ammp		2000
applu		1600
art	-startx 110	7500
crafty		700
eon	rushmeier	100
equake		21270
galgel		5000
gap		185330
gcc	166	2100
gzip	graphic	39300
mcf		12600
mesa		1300
mgrid		2100
parser		400
perl	makerand	10000
swim		1500
twolf		900
vortex	2	6000
vpr	route	36100

Table IV.7: Benchmarks used in Section IV.E

critical path predictor. In these runs, the latency of any instruction which was predicted to be critical was shortened by one cycle.

This type of evaluation produces two useful metrics, which we focus on: the fraction of all instructions predicted critical, and the fraction of possible speedup acheived from speeding up those predicted-critical instructions. (This type of evaluation cannot determine whether individual predictions are correct; we consider this issue in Chapter VI). The fraction of possible speedup is defined as t_{crit}/t_{all}, where t_{all} is the runtime when all instructions are shortened by one cycle and t_{crit} is the runtime when only predicted-critical instructions are shortened by one cycle. The possible speedup is defined as t_all/t_none, where t_none is the speedup when no instructions are shortened.

Table IV.7 shows the benchmarks used: 19 from Spec2000 Integer and FP. We perform all simulations using a detailed, execution-driven simulator, based on SMTSIM [113]. The simulator executes Alpha binaries, which are compiled with the DEC C (`-O4`) or Fortran (`-O5`) compiler. In all simulations, after advancing each thread to the simulation starting point indicated in Table VII.2 using a checkpoint, we performed a detailed simulation for 5×10^8 instructions.

Fetch up to 8 instructions per cycle
Branch prediction 64Kbit 2bcGskew, 4096 entry BTB
Pipeline 8 stage misp. penalty
Out-of-order execution with 96/96/96 entry integer/fp/memory instruction queues, which may issue 6 integer/mem instructions ($\leq$ 4 mem) and 3 fp instructions each cycle
Instruction Window supports 256 in-flight instructions
Memory system
16k 2-way 3 cycle L1 Instruction and Data caches
64 byte linesize
64 entry DTLB / 48 entry ITLB, fully associative
256 entry second level Data and Instruction TLBs
256k 4-way 14 cycle L2 cache
1MB 4-way 20 cycle L3 cache
100 cycle memory access time

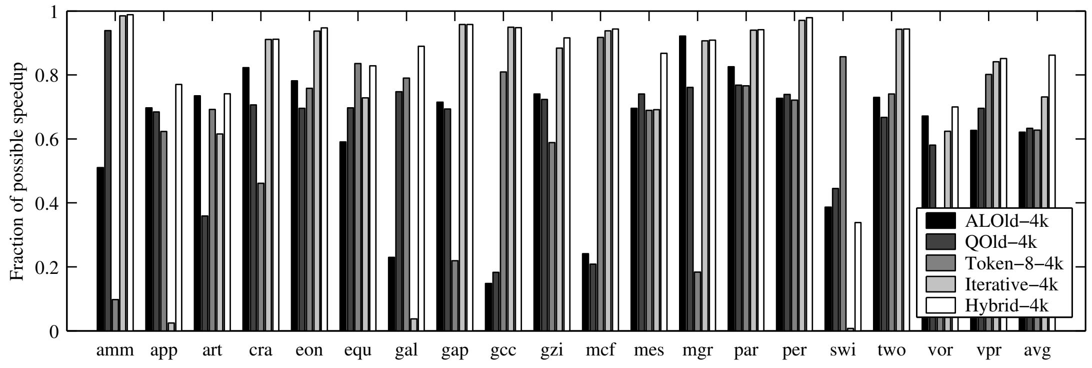
Table IV.8: Processor parameters for Section IV.E.

The parameters for the simulated processor are shown in Table IV.8.

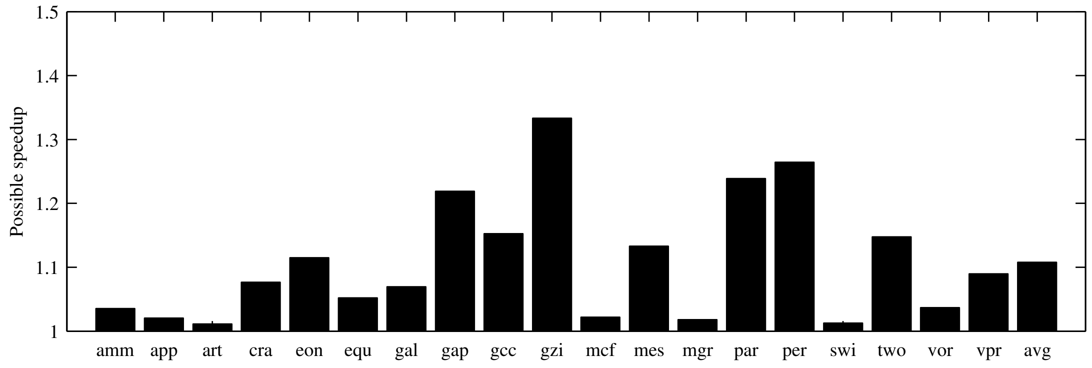
IV.E.2 Results

Figure IV.3(a) shows the fraction of possible speedup achieved by shortening all instructions which were predicted to be critical by a critical path predictor by one cycle. The legend shows the four types of critical path predictors evaluated. Of the critical-path predictors, the iterative predictor gets the greatest fraction of possible speedup, over all benchmarks. For reference, Figure IV.3(b) shows the maximum speedup from shortening instructions. It should be emphasized that *a greater speedup is not, by itself, indicative of a good critical path predictor*, in this evaluation. A predictor which predicts all instructions to be critical would get the maximum speedup, but would be of no use. An ideal predictor maximizes the speedup it gets from optimizing critical instructions, and minimizes the number of instructions which is predicts as critical. We can use the speedup as an indirect measure of the *coverage* of critical instructions: the fraction of all "actually critical"

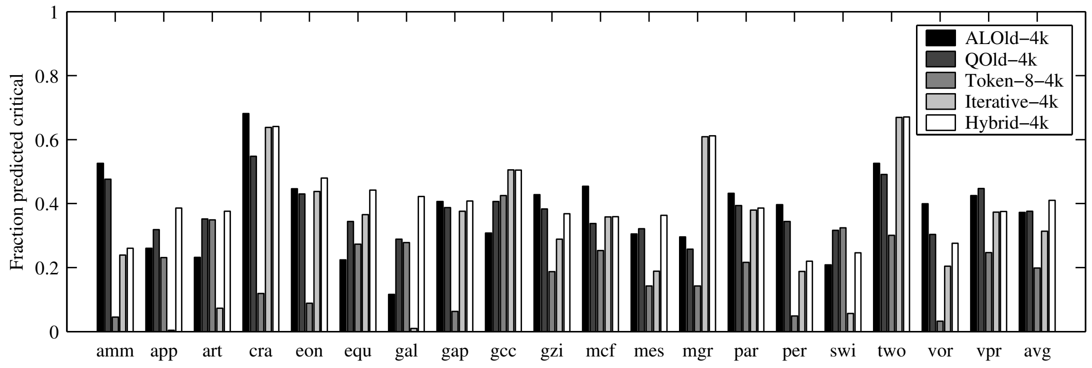
Figure IV.3(c) shows the percentage of instructions that were predicted to be critical by each predictor, and therefore, the number of times the latency of an instruction was shortened by 1 cycle. A lower number here is better. It



(a) Fraction of possible speedup achieved from shortening critical instructions



(b) Maximum possible speedup, measured by shortening all instructions



(c) Percent of instructions predicted critical

Figure IV.3: These three figures show the fraction of possible speedup achieved by shortening all instructions which were predicted to be critical by one cycle; the total possible speedup, measured by shortening all instructions by one cycle; and the fraction of instructions which were predicted to be critical by each predictor.

is a relative indicator of the *specificity* of the predictor–the fraction of predicted-critical instructions that are "actually critical".

IV.E.3 Analysis

Considering Figures IV.3(a) and IV.3(c), it is evident that the iterative predictor is Pareto optimal to the two heuristic predictors, because it gets more speedup from fewer instructions. The iterative predictor, hybrid and token-passing predictors are all Pareto optimal, according to this generic evaluation. Making further comparisons between predictors is difficult. The relative importance of coverage and specificity will depend on the application.

The hybrid predictor identifies the largest number of instructions as critical–about 40% on average. About twice as many as the token-passing predictor. But, it gets almost 90% of the possible speedup. The hybrid predictor concept appears to work quite well. For most benchmarks, the hybrid predictor operates largely in the iterative predictor mode. For the benchmarks `applu`, `galgel` and `swim`, where the iterative predictor fails, the hybrid predictor falls back onto a heuristic prediction. Consulting table IV.6, it can be seen that these three benchmarks all have low numbers of seeds. However, this alone is not enough to explain the failure of the iterative predictor, because `mesa`, for example, also has very few total seeds, but the iterative predictor gets nearly the same speedup as the heuristic or token-passing predictors. It is the combination of few seeds and the inability to grow a longer path from these seeds. Paths back from a seed may terminate early when, for example, they lead into a store whose value is not immediately used.

Interestingly, `mesa` also has few seed instructions, but the iterative predictor is able to get more of the potential speedup, perhaps because it can grow longer paths from each seed in that benchmark. As mentioned previously, both conditions A and B of the iterative predictor are quite strict. It may be possible to relax them slightly when very few seeds are being identified.

For some benchmarks, such as `gap` and `perl`, the token-passing predictor misses many critical instructions, as is evident in Figure IV.3(b). There are at least two reasons why the token passing predictor may perform poorly: the full-window criteria, and slow training.

The token-passing predictor assumes that the instruction window (or re-order buffer, or ROB) of the processor is the same size as the scheduling window (instruction queue). This is not the case with most real processors. Since the iterative predictor uses the criteria that the instruction window is full to determine criticality, it is more responsive to real hardware. The token-passing predictor might be modified to address this shortcoming, but we do not explore that.

The token-passing predictor may adapt more slowly to changes in program behavior, since it identifies critical instructions at a low rate. The token-passing procedure use to profile a single instruction can last over the period of up to 500 committed instructions, and only 8 instructions can be profiled at once (using the settings from [34].) Thus the token-passing predictor updates the CPB for as few as 1 in every 62 instructions, while the heuristic and iterative predictors train the CPB for every instruction.

It appears that much work is wasted by the token-passing predictor as each instruction is profiled. When the token passing predictor concludes that an instruction i is critical, it has passed the token over a span of 500 instructions. However, at that time the profiling reaches instruction $i+500$, when the profiling ends, the predictor only needs to wait a few more instruction, namely, to $i+n+501$, to see if a dependent of i, namely $i+n$, is also critical.

Finally, despite the fact that it cannot model all the nuances of processor hardware, the graph-model upon which the token-passing predictor is based still provides a good foundation for performing offline critical path analysis, which we do in the Chapter VI.

IV.E.4 CPB Size and Counter Format

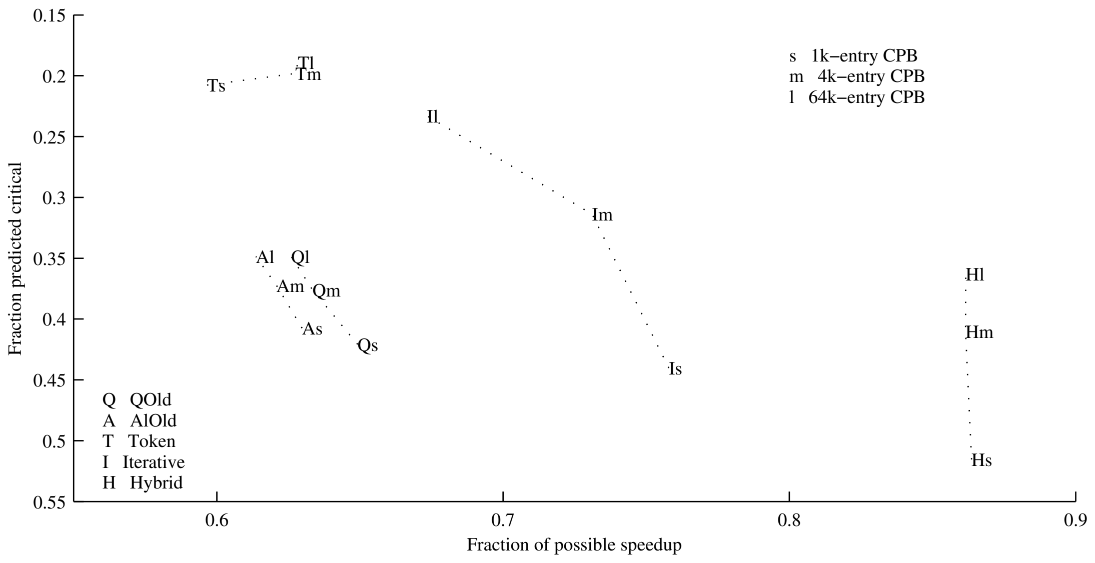
There are several parameters which specify the behavior of the CPB: the number of entries in the CPB, the amount by which counters are incremented or decremented, the maximum value at which the counter saturates, and the threshhold value above which instructions are critical.

For the experiments in the previous section, we used a 4k-entry CPB. Each counter: increments by 8, decrements by 1, saturates at 0 and 63, and predicts critical at 9 or above.

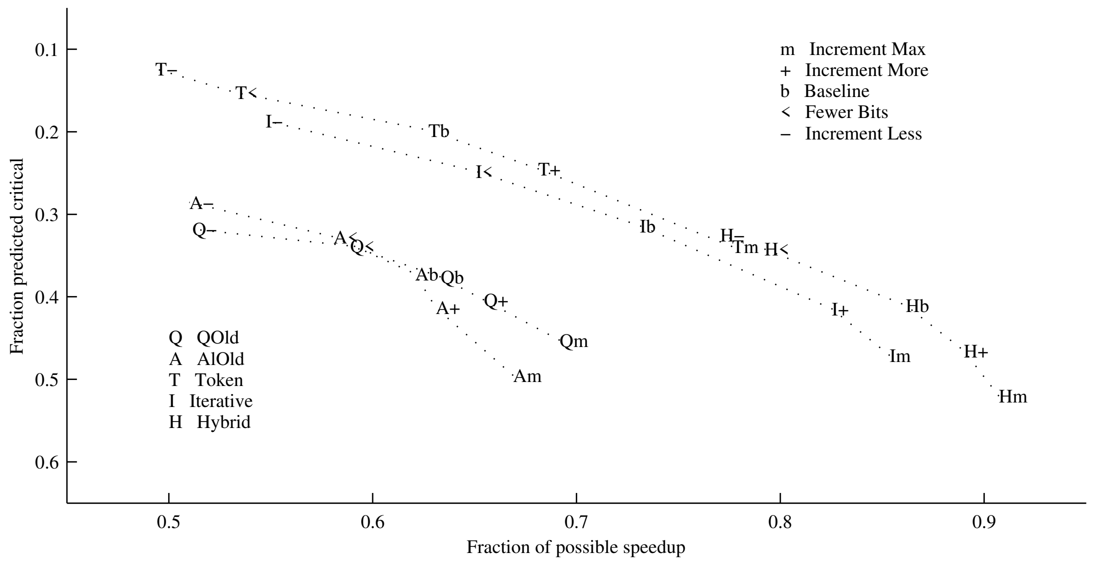
The total size of the CPB in bits is the product of the number of entries and the size of the counter in each entry. For the baseline predictor: 4k entries times 6 bits ($\log_2 64$) is 24kbits.

Figure IV.4 shows to pareto plots of the performance of different critical path predictors, each with different CPB parameters. The x-axis of each Pareto plot shows the fraction of possible speedup achieved. This is computed as described previously. Values further to the right are better. The y-axis of each Pareto plot shows the fraction of instructions predicted critical. This axis is inverted, so values toward the top are better. Points are labled with a two letter code, as indicted in each figure. The first letter of each code represent the predictor type, and the second represents the CPB configuration. The values for each label are an average over all benchmarks.

Figure IV.4(a) shows results for 3 different CPB sizes: 1k-entry, 4k-entry, and 64k-entry. At 6 bits per entry, this works out to 6, 24, and 384 kbits per predictor. Smaller tables will hold fewer recent predictions, and will have



(a) Fraction of possible speedup and fraction of instructions predicted critical, for different CPB sizes



(b) Fraction of possible speedup and fraction of instructions predicted critical, for different CPB counter policies

Figure IV.4: These two figures are show the fraction of possible speedup achieved versus the fraction of instructions predicted critical, for different CPB parameters. Speedup is due to shortening all instructions which were predicted to be critical by one cycle, and is expressed as a fraction of the speedup from shortening all instructions by one cycle. Points are labeled with a two letter code, as indicated in each figure. The values are an average over all benchmarks.

Description	Code	Increment	Decrement	Threshold	Maximum	Bits
Baseline	b	8	1	9	63	6
Increment More	+	12	1	9	63	6
Increment Less	-	4	1	9	63	6
Increment Max	m	63	1	9	63	6
Lower Threshhold	t	8	1	8	63	6
Fewer Bits	<	8	1	9	15	4

Table IV.9: Several CPB counter policies used in this study.

higher levels of aliasing. The token predictor is least sensitive to table size. For the other predictors, a smaller table increases the predict-critical rate, and the speedup. Because the counters are incremented more than they are decremented, any aliasing in the CPB tends to cause more instructions to be predicted critical. Some of these are, by chance, critical instructions which otherwise would not be predicted critical. For the hybrid predictor, it already gets nearly all critical instructions, leaving little room for improvement, hence the vertical orientation of its curve. The iterative predictor is most sensitive to table size. Because it builds on predictions to make new predictions, the effects of aliasing will tend to be magnified. (The iterative predictor might benefit especially from a few bits of tag for each CPB entry.)

Figure IV.4(b) shows results for 5 different CPB counter policies. These polices are shown in Table IV.9. As the figure shows, the predictors behave quite differently as the counters policy is varied. The variation is quite interesting in several ways. First, although each predictors behavior varys considerably over a large range, a certain overall ranking between the predictors is maintained. The results for the token and hybrid predictor together are pareto optimal to all the other predictors. The iterative predictor is close. The heuristics predictors are noticibly less optimal. (Recall that northeast is better, and southwest is worse in these plots.) This suggests that the counter policy varys the predictors behavior somewhat independently of the underlying accuracy of the identification mechanism. Because static instructions vary in their criticality, even a perfect identification mechanism cannot give perfect predictions. The counter policy can be used to trade off whether or not occasionally-critical instructions are predicted critical. Second, the relative ordering of the 5 policies along the dotted lines is the same for each type of predictor. This shows that the counter policies interact with all the predictors in the same fashion.

It should be pointed out again that these results are for a single, generic, optimization, which reduces the latency of instructions by one cycle. The optimal CPB configuration could be quite different for other applications. For example, reducing an instructions latency by one is less likely to change the critical path than reducing it by a large amount, or by value predicting it. For more aggressive optimizations, larger counter values may be needed to help *remember* critical instructions even when they are being optimized.

V

Critical Path Aware Optimizations

We have shown that it is possible to identify, in hardware, with good accuracy, which instructions are critical to the performance of a program. However, critical path prediction does not, by itself, make programs run faster. Instead, it is a *meta-optimization.* Predictions about which instructions are critical can be used anyplace that the processor needs to arbitrate between instructions. When critical instructions receive better treatment, performance increases. The previous chapter evaluated several critical path predictors in a generic manner. This section applies the predictions to more realistic optimizations. Table V.1 lists several critical path aware optimizations. Three are discussed in detail in this chapter, some have been evaluated by other researchers, and some are promising areas for additional work.

All these critical path aware optimizations share two important qualities: they involve a resource which is *optional* and *costly.* A processor resource is optional if it could be removed or ignored, and programs would still execute correctly, albeit more slowly. Predictors and caches are both examples of optional

Name	Description
Value Prediction	Value prediction is a performance enhancing mechanism. A limited rate at which instruction can be predicted presents an opportunity cost when selecting an instruction for prediction. Additionally, mispredictions create a direct cost. Both costs are mitigated by only value predicting critical instructions. See Section V.A, and in several papers: [19, 117, 34].
Instruction Steering	In a clustered processor architectures, instructions can, optionally, be assigned to execute in one of several locations. An additional execution latency may be incurred when dependent instructions are sent to different locations, but sending instructions to the same unit has an opportunity cost. The costs are mitigated when all or most of the additional latency is incurred by non-critical instructions. See Section V.B, and [117, 34].
Instruction Scheduling	When more instructions are ready to execute than there are execution units, critical instruction are given priority. Evaluated in [34, 32].
Multi-speed Functional Units	Power is conserved when non-critical instructions are executed by slower circuits. Discussed in Section V.C, and in [97].
Data Cache Tag-Check Serialization	Simultaneous tag and data lookup in a set-associative level-1 data cache consumes more power than a slower, serialized lookup. Non-critical instructions may tolerate this slower lookup.
Selection of Loads for Pre-Computation	Speculative threads can pre-compute loads which miss in cache, but forking such a thread has several costs. Limiting pre-computation to critical load misses mitigates these costs. Discussed in [22], and in Section VI.E.
Victim Cache	Several research groups, [37, 105], have proposed a victim cache with small blocks for data used by critical loads.

Table V.1: List of some critical-path aware optimizations

resources. The use of an optional resource may have a direct cost, such as power consumption, or it may have an opportunity cost—the use of that resource by one instruction prevents other instructions from using the resource. For example, a processor may have fast and slow integer execution units. Sending one instruction to the fast unit denies that unit to other instructions for at least a cycle.

Critical-path aware optimizations include selectively value-predicting critical instructions; steering and scheduling instructions in a clustered microarchitecture, or in a reduced-power microarchitecture with multi-speed functional units; and controlling how data is stored in a cache. And, while the cost of performing critical-path prediction is already modest, the cost is amortized when the predictions are used to control several optimizations.

The type of critical path predictor used and its application are largely independent. All three critical-path aware optimizations described in this chapter are evaluated using the heuristic critical path predictor. In each case, the performance would be even better if a more accurate predictor, such as the iterative predictor or token-passing predictor, were used.

This chapter is organized as follows: Section V.A describes *Critical Path Aware Value Prediction.* The performance of rate-limited value prediction can be improved if critical path prediction is used to select which instructions to predict. Section V.B describes a *Critical Path Aware Clustered Architecture.* Critical path predictions can be used to control the distribution of instructions to distant execution units in a way that minimizes the delays imposed by distance. Section V.C describes *Critical Path Aware Power Reduction.* Critical path predictions can be used to steer instructions to execution resources with speed and performance characteristics appropriate to the criticality of that instruction. Section V.D describes critical path aware optimizations which have been studied

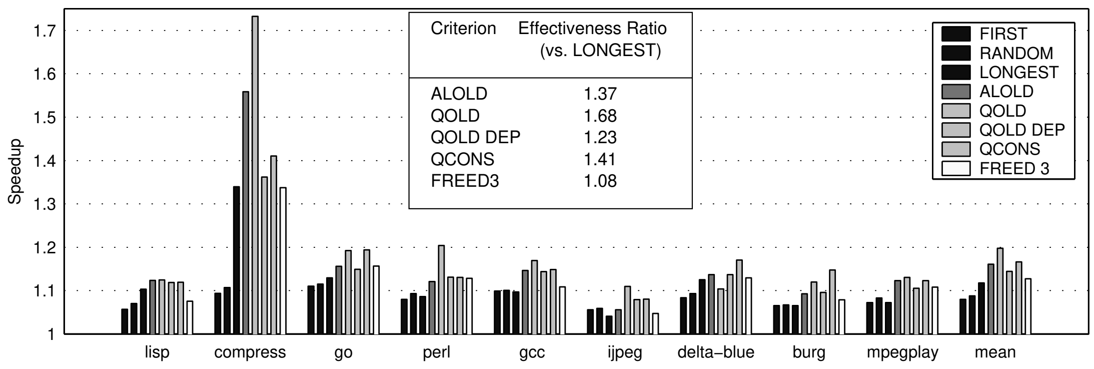


Figure V.1: The performance of value prediction incorporating critical path prediction.

by other researchers.

V.A Critical Path Aware Value Prediction

Value prediction is a microarchitectural technique which exploits value locality in programs by predicting the results of instructions before they execute. Value prediction can reduce the critical path by breaking dependence chains. Value prediction is described in Section II.A.

Calder et al. [19] studied ways to improve the performance of value prediction by controlling which instructions were predicted and which update the value history table. They allowed instructions which were on the longest path through the instruction window to be predicted if their predictor has a medium level of confidence, while other instructions were only predicted when the prediction carried a high level of confidence. This resulted in a better risk/reward tradeoff. The risk is a misspeculation recovery. The reward, increased performance, is more likely to result with instructions meeting the longest path condition. They also used the longest path condition to filter which instructions were allowed to update the value prediction table. In practice, however, keeping track of the lengths of paths in the instruction queue, and using that information immediately to control value prediction could prove difficult. Their paper did not suggest an implementation.

Critical-path prediction can assist value prediction in three ways. First, it allows the processor to make good choices when there are more predictable instructions in a fetch block than hardware resources to predict them. Second, it can be used to prevent costly misprediction penalties on instructions for which there is no benefit to prediction. Third, it can eliminate pollution in the value file by restricting which instructions are stored into it. Only the first benefit is examined in this chapter.

Any reasonable value predictor will have limited prediction bandwidth. Gabbay and Mendelson [40] showed that prediction bandwidth is important for the performance of value prediction. They developed architectures to provide multiple value predictions per cycle, but at the cost of increasing the complexity and access time of the value prediction architecture. We take the opposite approach. We attempt to achieve the maximum performance out of a value prediction architecture with a limited prediction rate (in this case, 1 per cycle) by using critical path predictions.

An instruction's predicted result needs to be written into a register so that dependent instructions can use the result. Multiple value predictions per cycle would require too many register file write ports. In some implementations of value prediction, a checkpoint needs to be recorded to allow recovery from an incorrect value prediction. Making multiple checkpoints per cycle may be impractical. Finally, a value history table capable of multiple predictions per cycle consumes more power, and occupies more chip area than a value history table which only makes one value prediction per cycle.

The results in this section were obtained using the experimental methodology described in Section IV.B.2. We use the heuristic critical path predictor scheme to evaluate this application, but the iterative or token predictors could easily be used as well.

Each cycle, value predictability information and, where applicable, dynamic critical path predictions are supplied for each instruction fetched. If multiple instructions are marked as value predictable, one of several heuristics is used to select one for prediction. These are described in Section IV.B.3 heuristics are similar to those already shown. FIRST and LONGEST select the first or longest-latency instruction, and RANDOM selects a random instruction. The remaining bars show the performance when using a CPB with the specified CP prediction criterion.

The results (Figure V.1) show that QCONS and QOLDDEP always provide more speedup than the selection schemes which do not use critical path predictions. QOLD delivers the best overall performance.

Because CP prediction is not an optimization, but an enabler for other optimizations, it needs to be evaluated differently. The absolute gains shown in this figure are strictly determined by the optimization we choose to model and the constraints we place on it. It is only the change in the optimization's effectiveness that is interesting. For that reason, we define the *Effectiveness Ratio* (ER) as follows:

$$\text{ER} = \frac{\text{Speedup}_{\text{with CP prediction}} - 1}{\text{Speedup}_{\text{without CP prediction}} - 1}$$

Therefore, if an optimization which provides a 20% speedup can achieve a 40% speedup when critical path prediction is incorporated, it has an effectiveness ratio of 2.0 – it has made the optimization twice as effective.

Viewed this way, QOLD achieves an effectiveness ratio of 2.26 over the RANDOM selector (it has made value prediction 126% more effective) and an effectiveness ratio of 1.68 over LONGEST. The speedup observed for `compress` is

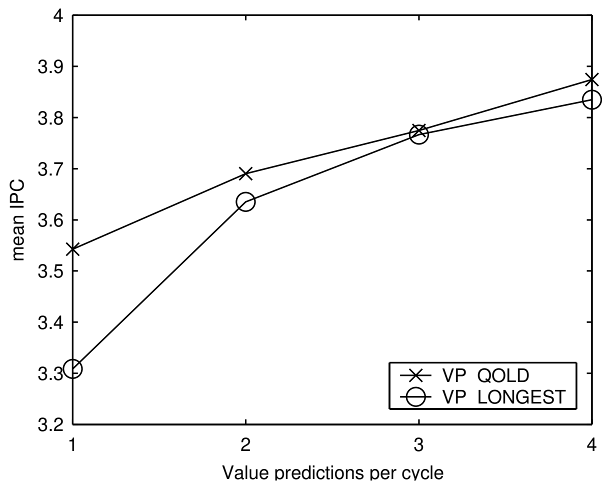


Figure V.2: The performance resulting from value-prediction of critical-path instructions, for varying value-prediction bandwidth.

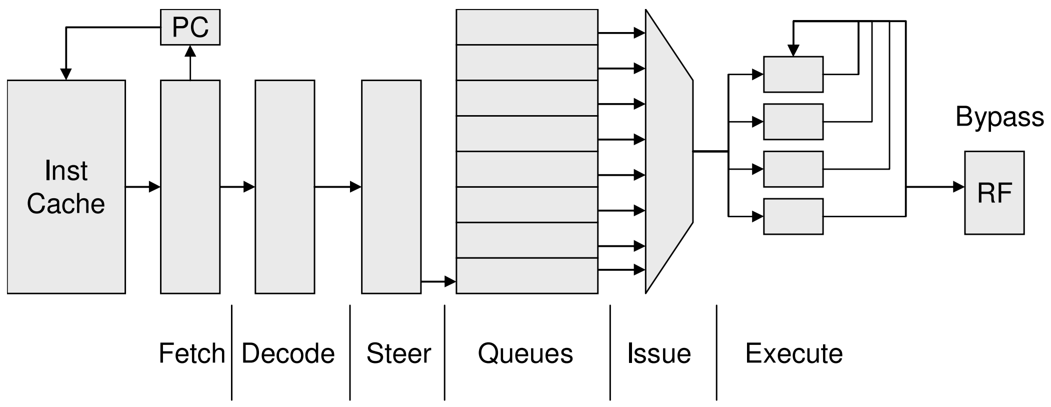
much higher than with the other benchmarks, but the technique is effective in all cases.

Determination of value predictability for these experiments is idealized to account for the continued improvement of those techniques and confidence estimators. In particular, we assume perfect value prediction confidence. Therefore, if the instruction would be correctly predicted by either conventional last-value techniques [71], stride techniques [39, 42], or a context-based predictor [92, 123], we mark it as value predictable. We simulate alias-free last-value and stride predictors. The context predictor is modeled after [123], with a 64K entry value history table, with four data values per entry.

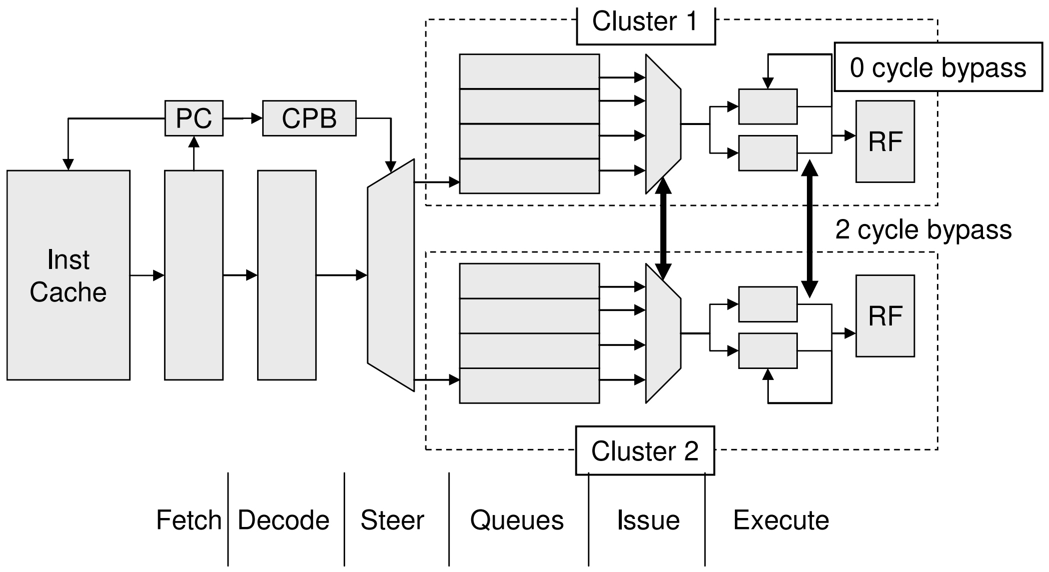
Figure V.2 shows the results of using a value predictor that can provide 1, 2, 3 and 4 predictions per cycle. The same benchmarks and simulator were used for this experiment as for the last. We have selected the best performing criterion from the previous experiment with one value prediction per cycle. Namely, the top line shows the mean speedup over all benchmarks for QOld. The lower line shows the speedup when the Longest selection scheme is used. The results show that the use of critical path information with 1 prediction per cycle bridges most of the gap between Longest with 1 and Longest with 2 predictions per cycle. With 2 predictions per cycle, critical path prediction still provides a noticeable increase over Longest. When more value predictions can be made per cycle, the two schemes start to converge, as the critical path arbitration becomes less necessary.

V.B Critical Path Aware Clustered Architecture

Clustered architectures reduce the complexity and delay associated with scheduling instructions and bypassing results. A conventional processor is redesigned as a clustered architecture by separating functional units and associated structures into multiple groups, or *clusters*. Figure V.3 shows a basic processor pipeline, and a clustered architecture. In a simple pipeline, as in Figure V.3(a), the issue logic or bypass logic may limit the cycle time of the processor. The capacitance of long wires within these structures is a source of delay. An alternative pipeline organization, is shown in Figure V.3(b). Decoded instructions are steered to one of two sub-pipelines, or clusters. Each cluster has its own instruction queue, issue logic, functional units, and bypass logic. Because the instruction queue for one cluster is smaller than in an unclustered processor, the delay of the issue logic is reduced. Because there are fewer functional units in a cluster, the complexity of the bypass logic is reduced. However, when dependent instructions are steered to different clusters, there may be a one or more cycle delay to receive a wakeup signal and/or to receive a bypassed result from the other cluster. Thus, compared to an unclustered architecture with the same total instruction queue capacity and number of functional units, a set of clusters can be operated at a higher frequency and could, in the ideal case, achieve the same instruction throughput, (IPC). However, in practice, due to the bypassing



(a) A simple processor pipeline.



(b) A clustered processor pipeline, with two clusters.

Figure V.3: Illustration of simple and clustered processor pipelines.

delay, actual throughput will typically fall short. This is an opportunity to use critical path prediction.

Performance on a clustered architecture is optimized when the instructions at both ends of key dependences are assigned to the same cluster. Even better, we'd like to send an entire critical dependence chain through a single cluster. Our approach is to always send predicted critical path instructions to the same cluster.

Variations on this basic clustered design are possible. The Alpha 21264 [60] has two clusters of integer functional units, served by a duplicated register file, but a single instruction queue. For the studies presented in this section, we simulate an architecture with two clusters of integer functional units, each served by a separate instruction queue. We assume bypassing of data between clusters takes 2 cycles longer than bypassing within a cluster. Instructions are assigned to a particular structure by hardware. This architecture is similar to that described in [59] and one of the machines described in [79]. A similar architecture is described by Farkas et. al., [31], but instruction scheduling is done statically. The M-machine [35] also features clusters, but their clusters are also not transparent to software.

The results in this section were obtained using the experimental methodology described in Section IV.B.2, except that the integer queue is divided in half, each serving half of the integer/load-store functional units. We will examine three different heuristics for assigning instructions to clusters, with increasing degree of complexity, and each being modified to incorporate critical path prediction.

The first technique, **Blind** assignment, assigns instructions randomly, with its only priority being to balance the load in each queue. **Blind_cp** sends all predicted-critical instructions to one cluster (if there is room); other instructions go to whichever cluster has more room. The blind algorithm suffers by not looking

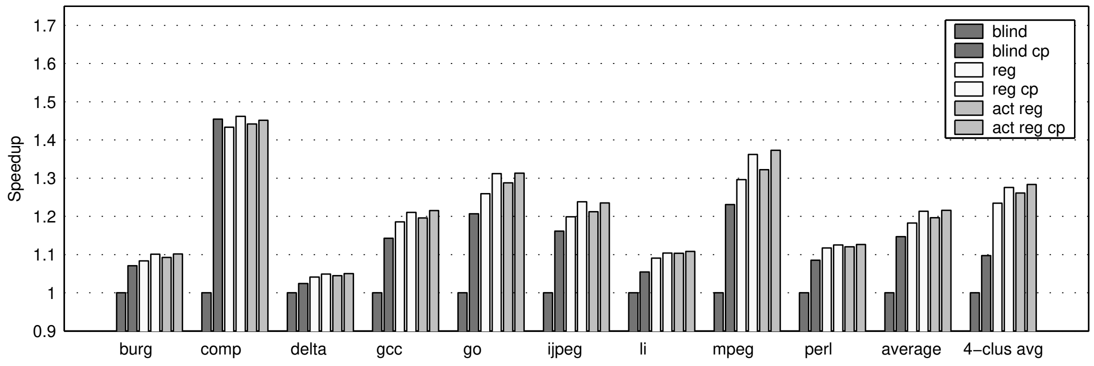


Figure V.4: The performance of a critical path-aware clustered architecture.

at register dependences, but has the advantage of allowing clustering to take place earlier in the pipeline, before such information is known, thus allowing more of the pipeline to benefit from decentralization.

The second technique, abbreviated **Reg**, takes register dependences into account. It attempts to send an instruction to whichever cluster the instructions providing the source operands were assigned. This is only violated when a queue is full or the queues are significantly out of balance. **Reg_cp** only uses critical path prediction to break ties when each operand comes from a different cluster.

The third technique, abbreviated **Act_reg**, is similar to *Reg*, but only considers the location of the producer of a source operand if that instruction has not yet completed execution (it is active). This is the mechanism closest to that assumed in [59] and [79], but is the most complex and assumes information not typically available to the early stages of the pipeline. **Act_reg_cp** again uses CP information to break ties when both operands are still waiting to execute.

A heuristic predictor (QOLD) was used for this application.

From Figure V.4, we see that the critical path prediction data allows better assignment of instructions for the less complex assignment schemes, achieving an average 15% increase over the *Blind* scheme, but a smaller gain over *Reg*. That gain is enough to allow *Reg_cp* to overtake *Act_reg*, possibly allowing a less costly way to achieve the result, particularly if we are already using the critical path predictor for other uses. With *Act_reg* we find even fewer ties that need to be broken, but the small improvement shown even there demonstrates that we are still making the right decisions when given the opportunity.

The last set of bars show the same results for a 4-cluster architecture. In that case we see that the blind allocation algorithm is more handicapped by the increase in clusters, but that the two register-based allocators are both more dependent on the critical path predictions to achieve their best performance.

V.C Critical Path Aware Power Reduction

Power consumption is an important design consideration for microprocessors. Many of the stages of the pipeline admit to multiple circuit designs, including the: instruction queues, functional units, instruction decoding circuits, and caches. Circuit changes which reduce power typically also reduce performance. However, much of this performance loss can be avoided by selectively reducing power consumption on per-instruction basis. Critical path prediction can be used to identify which instructions should be handled with reduced power.

This section summarizes the results of a study of using critical path information to mitigate performance losses resulting from power-reducing changes to two areas in the processor: the instruction queue and the functional units. To reduce power consumption in the functional units, we suggest that some functional units run at half the normal processor clock speed. To reduce the power consumption of the instruction queue, we suggest partitioning it, and making one half a simple fifo. Both these changes will reduce performance if applied indiscriminately. By sending non-critical instructions to the slower functional units, which they tolerate, and by sending critical instructions to the in-order queue, which they tolerate, the performance impact of the power-saving changes

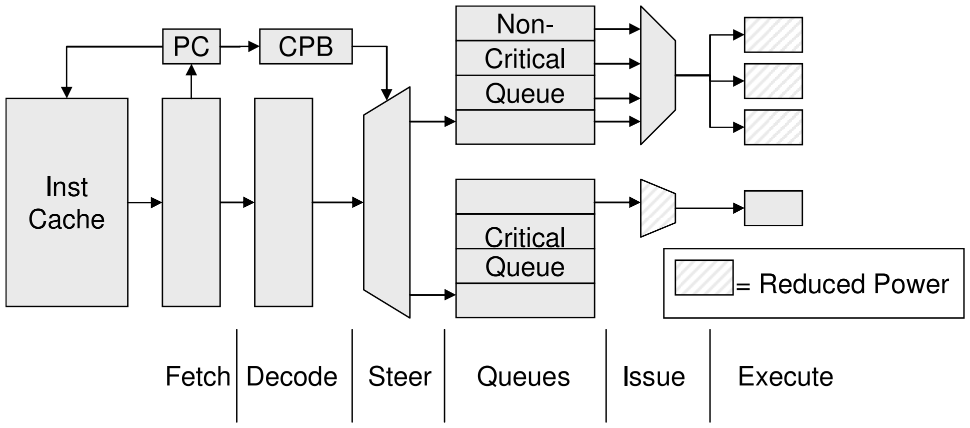


Figure V.5: A pipeline with critical-path aware power optimizations.

is reduced. Figure V.5 illustrates a processor pipeline with these changes.

Circuit blocks in general, and functional units, such as adders, in particular, can typically be implemented in several ways. Each design has different speed (performance) and power consumption. While the designer will only select a Pareto optimal design, he is still faced with many design alternatives. The same circuit can be designed with narrow devices, since the circuit need not operate as quickly. Also, a slower circuit with fewer devices can implement the same function. Static power consumption can also be reduced at a small cost in device speed. A better power/performance characteristic can be achieved if power decisions are made on a per-instruction basis. To accomplish this, the designer implements the same functionality in two different ways in the same processor. For the sake of discussion, call these two ways the *slow-cool* block and the *fast-hot* block. When there would be no performance difference, dynamic instructions should be executed using the slow-cool blocks. When there would be a performance difference, instructions should be executed using the fast-hot blocks. Critical path prediction provides the means to determine whether instructions should use faster functional blocks or a lower-power blocks.

V.C.1 Approach

In [97], we presented a microprocessor architecture which uses critical path prediction to achieve an improved power/performance characteristic and reduced power density in potential hot-spots. Seng was the primary author and researcher of that work, and it also appears in his thesis, [96]. Tune was a secondary author and researcher. However, we do present some of the results of that study here. Like other works discussed in this chapter, it represents an important application of critical path prediction, and a validation of the prediction techniques developed in this thesis.

Again, Figure V.5 illustrates a processor pipeline with critical-path aware power optimizations. As each block of instructions is fetched, the same addresses are sent to the critical-path prediction buffer (CPB). By the time the instructions complete the decode and rename stages, the CPB responds with a prediction of the criticality of each instruction. This prediction is then used to steer the instructions to either a fast or slow side of the execution pipeline. At the same time, as instructions finish execution, the critical path predictor is retrained. For this work, the QOLD critical-path identification heuristic, described in Chapter IV, was used.

The slow side of the pipeline is an out-of-order instruction queue which issues to multiple slow, lower-power functional units. The fast side of the pipeline is an in-order instructions queue which issues to one fast functional unit. Predictions from the CPB are used to steer decoded instructions to the appropriate half of the pipeline.

The technique of slotting instructions before dispatch into an instruction queue was used in the Alpha 21264 [60]. Once an instruction is slotted to the fast or slow side, it must remain on that side. Normally, instructions predicted as critical are steered to the fast side of the pipeline, and instructions predicted

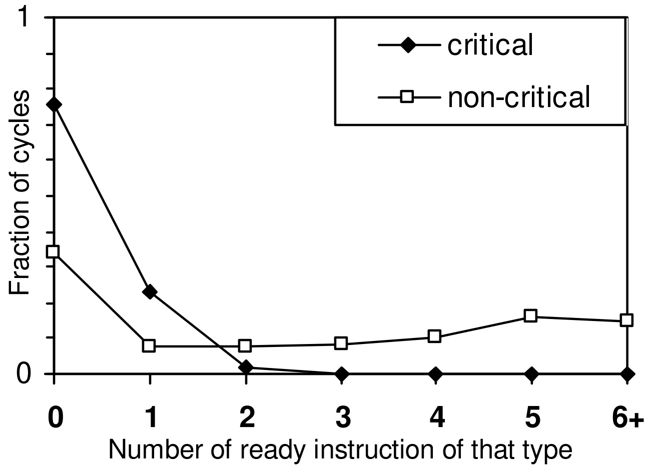


Figure V.6: Histogram of the distribution of the number of ready-to-execute instructions which were predicted to be critical or non-critical on a particular cycle. From [97]. Using a single 64-entry instruction queue.

to be non-critical are slotted to the slow side of the pipeline. However, if the ratio of critical to non-critical predictions does not match the throughputs of the two pipeline sides, then the instruction queue on one side will fill up while the other goes unused. To avoid such a condition, load-balancing logic can override the critical-path prediction, and send instructions to the underutilized side of the pipeline.

The critical path will include many chains of data dependent instructions, and the critical path predictors described previously will tend to identify chains of dependent instructions. It is thus reasonable to expect, that, in most cases, that only one critical instruction will be ready on a given cycle. Figure V.6 shows the number of predicted-critical and predicted-non-critical instructions ready to execute each cycle in a 64-entry instruction queue, from [97]. As the figure shows, there are rarely two critical instructions ready on the same cycle. To take advantage of this, the queue for the non-critical instructions can be modified to only consider the one instruction at the head of the queue for scheduling each cycle.

In some cases, an instruction may be critical when executed on the slow unit, and non-critical when executed on a fast unit. An offline analysis, as in Chapter VI or [32], can be used to compute how much slack, in cycles, each instruction has. However, it would require considerable hardware to compute a numerical value for slack at runtime. Instead, because the counters of the Critical Path Buffer have hysteresis, slow path will end up being treated as critical. This approach favors performance over power when there is an unstable situation. This is the approach taken here, and extended in [32] to include different predictions about the criticality of an instruction with respect to multiple resources.

V.C.2 Evaluation

We summarize here the how we modeled power consumption in that paper. Additional background on power-reduction can be found in Section II.B. To evaluate critical-path aware power reduction, we combined a detailed, execution based simulation of a processor and an architectural power model. The Wattch power model [15] measures dynamic power consumption as a function of the fraction of cycles on which a block is used, and empirically determined constants. Static power consumption of a unit in the baseline processor was assumed to be 10% of of the dynamic power consumption of that unit when active, following [109]. Dynamic power consumption of the slower, lower-power blocks is modeled as 80% of that of the corresponding faster block, and static power consumption is assumed to be 50% of that in a faster block.

Figure V.7, adapted from [97], evaluates several different processor designs with and without critical path power optimizations. The designs are evaluated with performance divided by power as a figure of merit. The y-axis shows the ratio of performance to power. Performance is measured in instructions per cycle. The sum of the instruction queue and functional unit power is measured

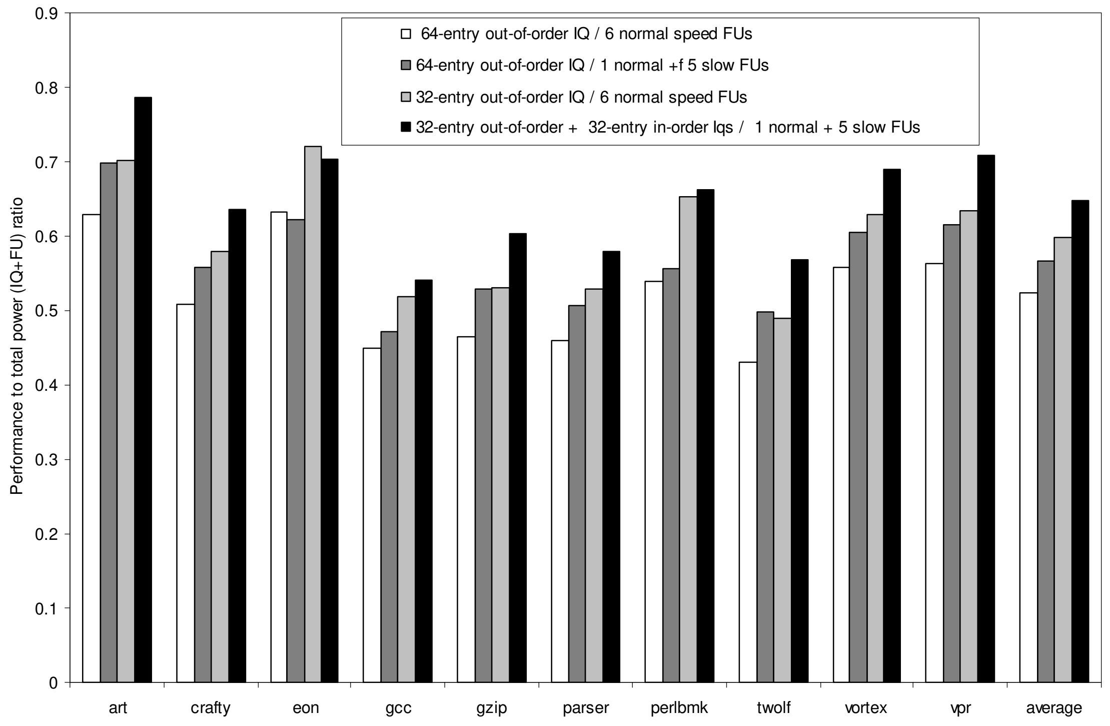


Figure V.7: The effect of different instruction queue sizes and arrangements, and different functional unit speeds, on power and performance. From [97].

in arbitrary units. The leftmost bar in each group represents a baseline configuration with no power optimizations, and no critical path predictions being used. Proceeding rightwards in the bar-group, the next bar represents a configuration with multi-speed functional units. As indicated in the legend, there is one normal speed functional unit, and 5 slow functional units. For comparison, the baseline has 6 normal speed functional units. For most benchmarks, this configuration has a better figure of merit. The next bar shows that simply reducing the size of the instruction queue from 64 entries to 32-entries also reduced power consumption more than it reduces performance. The leftmost bar represents a configuration which uses both multi-speed functional units, and two instruction queues, where one instruction queue is in-order, and thus has a reduced power consumption. Critical path information is used to steer instructions to the appropriate queue and functional unit group. This configuration has the highest figure of merit, on average. On average, this configuration, which has a 32-entry in-order queue with five slow functional units and a 32-entry out-of-order queue with one normal functional unit, shows a 20% improvement in the power-performance metric over the baseline design, which has a unified 64-entry queue and 6 fast functional units. The benchmark `eon` stands out from the others. It does not benefit from the critical-path aware optimizations. A more accurate critical-path predictor could ameliorate this.

Again, these results are for the power consumed by the instruction queue and the functional units. Estimates of the contributions of these portions of the processor to overall power consumption vary considerably. Various sources [38, 2, 15] claim that the power consumption of integer and floating point units is between 2% and 22%, and that of the instruction queue is between 8% and 27% of total processor power consumption.

Thus, according to some sources, the functional units and instruction queue may, together, account for between a half and a third of the total power consumed by a microprocessor. However, if these are areas of high power density, [48], then reduction of power consumption in these areas alone may be a worthwhile goal in itself.

Also, the additional power consumption from the critical path predictor was not considered in this study. However, the power consumption of the critical path predictor itself can be mitigated in several ways. Its size can be made quite small–less than 1k entries–for many programs before its effectiveness degrades. An access to the CPB can take multiple cycles (if pipelined) since the predictions are not needed until several cycles after the fetch addresses are known. Further, the storage elements in the CPB can be implemented with weaker, error-prone, storage elements because prediction errors do not affect correct execution. Finally, if the critical path predictor is used for other purposes as well, its cost can be amortized.

V.D Other Critical Path Aware Optimizations

Bahar et al. [7], classified instruction cache misses as being critical or non-critical, and excluded non-critical instruction cache lines from the cache, which resulted in an energy savings. Fisk and Bahar [37] classified data cache lines as critical or non-critical, and excluded non-critical lines from the data cache, which reduced conflicts, and improved performance.

Fields et al. [34] studied both critical-path aware value prediction and critical-path aware clustered architectures. They used criticality information to mitigate the cost of value mispredictions. They found up to a 5% performance improvement from using critical path information. In addition to using critical path information to steer instructions in a clustered architecture, as we did, they also gave critical instructions priority when there was contention for functional units. They found up to a 15% performance improvement from using critical path information to schedule instructions, and up to a 20% performance improvement when also using critical path information to steer instructions as well. In [32], they studied using critical path prediction for power reduction. They considered a similar architecture, with two pipelines, side by side, with one clocked at half the frequency of the other. They used their critical path predictor to steer instructions to the slow or fast pipeline, and to prioritize instructions for issue within the scheduling queues of each pipeline half. They found that there was a 1-2% loss in performance over a configuration with all full-speed pipelines, which suggests the potential for considerable power savings, although they did not incorporate a power model into their simulations.

Srinivasan et al. [105] proposed a method for identifying and predicting critical loads, which was described in Section IV.D. They apply these predictions to two memory hierarchy optimizations. They study a victim cache, [56], that holds only lines touched by critical loads. The criticality-based filtering may reduce contention in the victim buffer. They also investigate a form of prefetching limited to data associated with critical loads with the aim of reducing bus contention. However, they found that, while only some loads may be critical, their associated data sets are disproportionately large.

Racvik et al. [87] proposed a method for identifying non-critical loads, which is described in Section IV.D. They propose a level 0 cache which holds only data associated with critical loads.

Collins, [22], used a modified version of on of the heuristic predictors described in Chapter IV to select loads to speculatively pre-compute.

Many other applications of critical-path prediction are waiting to be evaluated. We suggest two more:

Multiple-path execution [122, 47, 63] follows both targets of conditional branches that have low prediction confidence. Better use of prediction resources could be obtained by not forking non-critical-path branches, or perhaps not forking branch directions that are not immediately on the critical path.

Normally, accesses to the L1 data cache tag array and data array are done in parallel. This speeds up the data cache access, but, for a set associative cache, does consume excess power, since data must be driven out of all ways, rather than just one. For non-critical loads, tag and data access could be serialized. Other researchers have considered using way-prediction to reduce data-cache access energy [84].

VI

Quantifying the Critical Path

This chapter seeks to increase our understanding of the dynamic critical path and critical path predictors in several ways. All the critical-path predictors discussed in Chapter IV produce a binary classification of criticality. In this chapter, we seek to assign a value to the criticality of instructions–one which denotes the amount of benefit available from optimizing an instruction. We show the distribution of criticality and the variability of criticality for several programs. The critical path prediction buffer which we describe in Chapter IV predicts instruction criticality based on the program counter. However, in this chapter, we show that criticality is quite dynamic for many instructions that are actually on the critical path at least some of the time. Thus, PC-based predictors are limited in how well they can predict the critical path, even with very accurate training information. This chapter examines whether instruction criticality is more highly correlated to other index functions which might include information such as criticality pattern history, branch history, or load history. In particular, we show that a critical path predictor which uses the local history pattern of criticality can significantly improve critical path prediction accuracy.

This chapter presents an offline critical path analysis: a framework that identifies for each dynamic instruction both whether, and to what extent, it is critical. This technique is computationally intensive, and is not intended as another dynamic critical path predictor. Rather, it is a tool for understanding how dynamic instructions differ in their impact on program runtime. Using this offline analysis, we: i) evaluate the quality of predictions for several proposed critical path predictors, ii) study the correlation between the criticality of dynamic instructions and the corresponding static instructions, iii) correlate criticality with other events (e.g., branch history and load history) in the pipeline, iv) measure the slack (distance from being critical) present in non-critical instructions, v) present a definition of *tautness*, a quantification of the importance of critical instructions with respect to optimization, and vi) present the distribution of slack and tautness among instructions.

The rest of the chapter is organized as follows: Section VI.A describes the simulator and the benchmarks used in this study. Section VI.B describes our approach to quantifying the criticality of instructions. Section VI.C evaluates several different proposed critical path predictors. Section VI.D studies the distribution of critical and non-critical instructions in programs. Section VIII.A.3 concludes.

VI.A Methodology

Our framework for this research consists of three parts: a detailed simulator that produces a dependency trace for each application, a directed graph of the dependencies in the program built from this trace, and a program called the *rescheduler* which computes the effect on the execution time of the program as various dependencies are changed. The graph model is based on that proposed by Fields. It is described in detail in Section III.E. In this section, we describe the simulator and the set of benchmarks we use to generate the dependency trace and again to validate the results of the rescheduler. In the next section, we describe

Parameter	Value
Fetch width	2 basic blocks/cycle 8 instructions/cycle
Issue width	8 instructions/cycle
Commit width	8 instructions/cycle
Branch Predictor	8k/8k-entry local-history, 16k-entry global, 16k-entry choice 8-cycle mispredict penalty
L1 Data Cache	16kB 2-way (8-cycle miss penalty)
L1 Inst Cache	16kB 2-way (8-cycle miss penalty)
L2 Cache	256kB 4-way (20-cycle miss penalty)
L3 Cache	1MB 4-way (100-cycle miss penalty)

Table VI.1: Processor parameters used in this study.

the rescheduler and the constraint-graph model.

Simulations are performed using a detailed architectural simulation of an out-of-order processor executing the Alpha instruction set architecture. Simulations for this research were performed with the SMTSIM simulator [113], used in single-thread mode. The parameters for the processor are summarized in Table VI.1. The simulated processor has a reorder buffer of 255 instructions. Our simulated processor does not have a limited instruction queue; it is only limited by the size of the reorder buffer. We made this concession to realistic processor modeling because the graph-model does capture the effect of a limited instruction queue. The processor can fetch, execute, and commit up to 8 instructions per cycle. It can fetch up to two non-contiguous basic blocks per cycle. The memory system models contention at each level of the memory hierarchy.

We chose 5 SpecFP2000 and 8 SpecINT2000 benchmarks for this study. The benchmarks were fast forwarded (emulated but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation began. Then, the cache and branch predictors were warmed up for 50 million instructions for all benchmarks. Finally, the critical path measurements are based on 10 million instruction-long traces after warmup. The benchmarks used, their inputs, and the number of instructions fast-forwarded, are shown in Table VI.2. The reference input was used for all benchmarks, and where there are multiple

Benchmark	Code	Input	Fast Forward Instruction ($\times 10^6$)
Floating Point			
ammp	amm	ref	2700
applu	apl	ref	500
equake	equ	ref	3000
galgel	gal	ref	2600
swim	swi	ref	800
Integer			
crafty	cra	ref	1000
eon	eok	ref (kajiya)	100
gap	gap	ref	1000
gcc	gc2	ref (200)	10
gzip	gzp	ref (program)	50
parser	par	ref	320
twolf	two	ref	2500
vpr	vpr	ref	1000

Table VI.2: The benchmarks used in this study.

reference inputs, the one used is indicated.

VI.B Measuring Critical and Slackful Instructions

Briefly, the critical path is the longest path through a weighted directed acyclic graph which represents the ordering of events and duration of activities. Section III.A provides general background on critical-path analysis. A program executing on a processor can, to a significant extent, be modeled by such a graph. Many types of dependences and constraints can be modeled with graph edges. Fields et. al. propose such a graphical model in [34], which we summarize in Section III.E. The longest path length of this graph corresponds to the execution time of the program. Fields et al. classify instructions as being "fetch critical", "execution critical", or "commit critical" if delaying the fetch, execution, or committing of the instruction would delay the overall program's execution. In this chapter, we are only interested in whether instructions are "execution critical". We focus on this definition of criticality because all critical-path aware optimizations studied thus far work by hastening or delaying the execution of instructions.

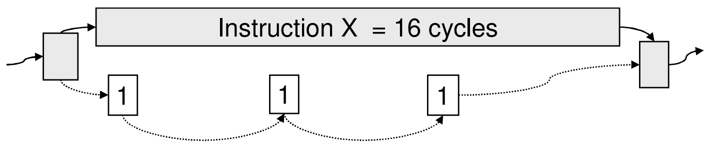
VI.B.1 Slack and Tautness

This chapter focuses on two metrics, slack and tautness, to quantify instruction criticality. Intuitively, slack represents how far an instruction is from becoming critical. *The slack of an instruction is the number of cycles that the instruction can be delayed without increasing the execution time of the program.* Instructions with more than zero cycles of slack are non-critical.

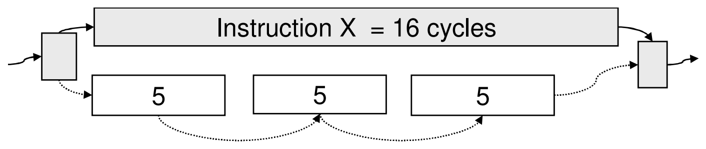
We propose a new metric, *tautness*, for distinguishing critical instructions, which corresponds to how far away an instruction is from becoming non-critical. Tautness is a complementary measurement to slack, for instructions which are critical. *We define the tautness for an instruction as the number of cycles by which execution time is reduced when the result of that instruction was made available to other instructions immediately.* For all instructions that write a result to a register, this means making that result available as soon as the producing instruction is dispatched. For store instructions, this includes making the value stored available to dependent loads. For mispredicted branches, this means removing the misprediction.

Figure VI.1 illustrates tautness. Two fragments of a dataflow graph are shown, where the length of the boxes signify the latency of an instruction. Non-critical instructions and dependencies are shown with lighter boxes, and dotted lines, respectively. In both diagrams, instruction X has a latency of 16 cycles, and is on the critical path. In Figure VI.1(a), X has a tautness of 13 cycles, because the next longest path when X is removed is 13 cycles shorter. In Figure VI.1(b), X has a tautness of 1 cycle, because the next longest path is only 1 cycle shorter.

Tautness is a useful measurement because it quantifies the maximum benefit of applying an optimization to an instruction. It roughly models what might be achieved by value predicting or speculatively pre-computing the result of an instruction. Notice that this information is not immediately available by



(a) Critical instruction X has tautness of 13 cycles.



(b) Critical instruction X has tautness of 1 cycle.

Figure VI.1: An example illustrating tautness.

identifying or analyzing the longest path through the graph. For example, an instruction with a latency of 100 cycles would thus contribute 100 cycles to the length of the longest path, but removing that instruction from the program graph might expose another path whose total length is only 1 cycle shorter than the original path. In that case, the instruction has a tautness of 1 cycle. Tautness accounts for all paths through the program, not just the longest.

The design of an implementable critical path predictor that returns a tautness value is left to future work, but such a predictor would have several benefits. (1) If the critical path predictor is used to arbitrate a constrained resource, a binary critical path predictor cannot distinguish between multiple critical instructions which want to use the resource. (2) Some optimizations, such as speculative pre-computation [23, 22], devote significant resources to target a single instruction. In those cases, it is not sufficient to target critical instructions, but rather we would only want to target critical-path instructions that exceeded a tautness threshold. Speculative pre-computation [23] could use a static critical path predictor that included tautness (that might look something like our rescheduler), but dynamic speculative pre-computation [22] would require a dynamic predictor.

Our critical-path analysis framework allows us to precisely measure these two properties of instructions (slack and tautness). We first discuss the constraint-graph model of the critical path that we base our work on, and the algorithms we used to extend the constraint-graph model and to compute slack and tautness.

VI.B.2 The Rescheduler

In [34], Fields et al. used the graph model to determine what instructions were critical in a particular execution of a program. This graph is described in detail in Section III.E. In this chapter, we use the graph-model to efficiently determine the effect on the execution time of the program if each instruction were executed sooner or later. We use the *rescheduler* to efficiently determine the effects of changes to a large graph. We also use the rescheduler to model a processor constraint, limited issue bandwidth, which cannot be represented in the graph.

The simulator which is used to generate program traces is described in Section VI.A. A program trace provides information about each dynamic instruction, including fetch delays, execution latency, execution dependencies, and branch mispredictions. Using the rescheduler, which takes this trace as input, and converts it into the directed graph model of [34], we can compute the effect of making an instruction complete execution earlier or later than it did in the original simulation. The rescheduler can compute the effect of changing a dependence faster than rerunning a simulation, and thus it is practical for us to make separate measurement for each instruction in a program trace containing tens of millions of instructions.

VI.B.3 Using Rescheduler to Measure Slack and Tautness

We use the rescheduler to compute two metrics for each instruction: *slack* and *tautness*, as defined earlier. The rescheduler operates on a moving window of the graph, since the entire graph would be too large to store efficiently in memory. The longest path to each node is computed for all nodes, which are already in topologically sorted order as generated from the initial simulator trace. Next, to compute the effect of a change in the graph, the graph is changed as desired, and the longest path is recomputed for all nodes that follow the changes. However, it is not necessary to recompute the longest path for the entire graph each time a node is changed in order to determine the total effect on the program execution time. We exploit the fact that no edge spanning more than R instructions is ever on the longest path, where R is the size of the instruction window. As the longest path to each node is recomputed, the change in the time between the original schedule and the modified schedule is computed. When this difference, Δ, is constant for a run of consecutive instructions of length R, then we can say with certainty that all subsequent nodes in the graph will also change by Δ; thus, the runtime of the program changes by Δ. The rescheduler is feasible, even as an offline technique, only because changes to the constraint graph always have a localized effect on the entire graph.

To compute the tautness for an instruction I, the rescheduler removes any data-dependence edges out of I's E-node. This allows instructions that depend on the result of I to execute independently of I. However, I must still execute eventually. Thus, if I is "commit critical"–it causes the instruction window to fill up when a critical instruction is just outside the window–then I will have a tautness of 0. We chose to define tautness this way so as to be similar to optimizations such as value prediction, where the instruction that is the target of optimization must still execute to validate speculative data.

To compute the slack for an instruction I, we delay the execution of I by a large number of cycles, and recompute the longest path for the graph. The difference between the amount by which I was delayed and the increase in the longest path is the slack in executing instruction I. For example, if delaying instruction I by 100 cycles causes the program to run 98 cycles longer, we conclude that I has 2 cycles of slack, after which it became critical. Using this graph-adjustment approach, we compute the slack and the tautness for each e-node in the graph (every dynamic instruction in the program trace.) While there are are more efficient algorithms for computing slack in an ordinary graph, the additional issue-width constraint prevents the direct application of such an algorithm in this case.

We also augmented the longest path computation to adjust for the effect of a finite number of functional units. For the execution-node, e, of instruction I, the longest path to node e would correspond to the cycle at which I executes, *if* there was not an issue limit. To model this additional constraint, after computing the longest path $l(e)$ to a node e, we consult a table to see how many older instructions were scheduled for the same cycle. If all functional units are already busy at cycle $l(e)$, then we increase $l(e)$ until a issue slot is found. This works because we assume that the hardware scheduler gives preference to older instructions when picking from among all of the available instructions that are ready to execute. This is only one of the resource constraints previous graph-based approaches do not handle well, but it serves as an example of how others could be handled, such as more specific functional unit constraints (load-store units, dividers) or a limited size instruction queue. These limitations are handled by a combination of the constraint graph and a mechanism for processing the graph.

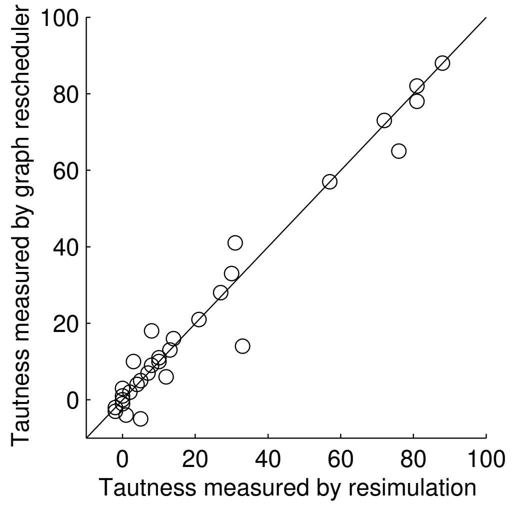
In the original graph-model, [34], the cycles that an instruction spends ready and waiting in the queue due to functional unit contention are included in the weight of the EE and EC edges. This is sufficient for determining whether an instruction was critical in the unchanged graph. When we model the token passing predictor of [34] in this chapter, we include contention cycles in this fashion. But that is not adequate for our purposes. We are interested in the effect of changing the graph. Changing the graph also changes the conflicts. Therefore, we ignore any contention present in the program trace, and recompute the effect of contention in the rescheduler.

VI.B.4 Validation of the Rescheduler

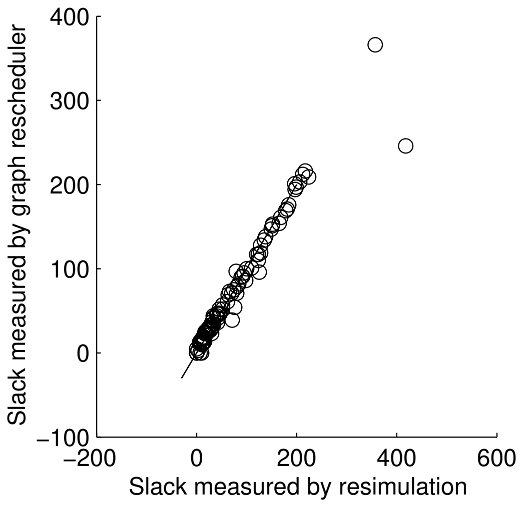
The rescheduler and the dependence graph together incorporate many but not all of the effects modeled by a full simulation. Both the rescheduler and the simulator compute the cycle when each instruction is executed, and the total number of cycles to execute the program. There is a certain amount of error in the cycle times computed by the rescheduler due to effects not modeled.

One way to measure this error is to compare the cycle at which each instruction executes in the simulator, and when it executes in the rescheduler. However, this sort of validation can only test whether the unchanged graph is correct. This was the approach taken in previous chapters, and in prior work. Since we are interested in changing the graph, it is important that the graph remains meaningful in the face of changes. The property of remaining accurate in the face of changes may be termed *robustness*.

In order to validate using the rescheduler/graph to calculate slack and tautness, we randomly selected dynamic instructions with a range of different slack and tautness values, and then compare the slack or tautness computed by the rescheduler/graph with their corresponding values from the detailed simulator. To measure slack and tautness in the simulator, we ran the simulator with that one dynamic instance of the instruction delayed (to measure slack) or with



(a) Tautness



(b) Slack

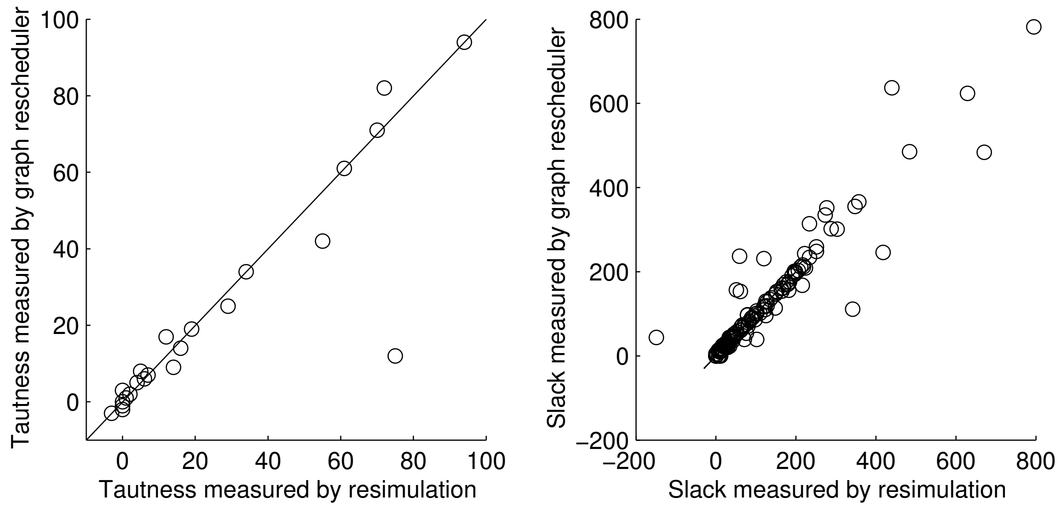
Figure VI.2: Comparison of (a) slack and of (b) tautness values, as computed by the rescheduler and as determined by re-simulation, for `gcc`.

that one instruction's result available early (to measure tautness)[1].

Figures VI.2(b) and VI.3(b) are scatter plots showing the agreement between the slack measured by the simulator and the slack measured by the rescheduler, for a random selection of instructions that our technique indicates has slack. Figures VI.2(a) and VI.3(a) shows the same type of scatter plot, but for tautness.

We validated the rescheduler on a range of benchmarks. We present the results here for `twolf` and `gcc`, because those results fell in the middle of the benchmarks measured — some correlated better, some worse. The line $x = y$ is drawn for convenience. Points that lie on this line represent instructions where the rescheduler agreed exactly with the results of re-simulation. For most

[1]The instructions result is made available as soon as it is dispatched, which reflects the performance improvement a program would enjoy if the instruction in question were correctly value predicted. An alternative definition of tautness, which we call effective tautness, is if the instruction has zero latency, buy must still wait for both its operands to be ready before it produces a result. The effective tautness cannot be greater than the latency of the instruction, even if it lies on a strongly critical path, and can be calculated from tautness.



(a) Tautness (b) Slack

Figure VI.3: Comparison of (a) slack and of (b) tautness values, as computed by the rescheduler and as determined by re-simulation, for `twolf`.

instructions, the tautness (or slack) measured by the rescheduler is at or very close to the result obtained from a full detailed simulation run. The next section discusses some of the reasons why the correlation is not perfect.

Sources of Error in the Model

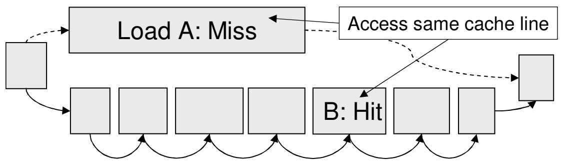
We use the graph/rescheduler to measure slack and tautness faster than would be possible through re-simulation. The constraint graph is, however, a simplification of all the interactions that take place in a real processor, and some inaccuracies will result.

One type of discrepancy occurs because the memory hierarchy is only modeled indirectly in the graph. Load instructions have a variable latency. The latency on data-dependence edges associated with a load instruction are the actual execution latency of the load during the initial simulation. Once the weights are assigned, they do not change. In the most common case, the execution latency of a load instruction is the same regardless of changes to the graph. That is, in the common case, the latency of a load is independent of when it, and other instructions, execute.

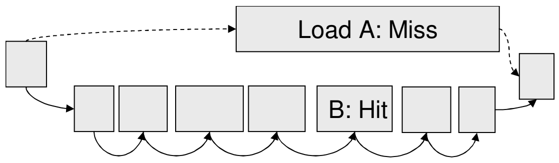
However, there are three ways in which memory latencies can change that are not modeled by the rescheduler. First, the simulator models conflicts between non-dependent instructions for cache banks and data buses, but the rescheduler does not. Second, changes to load ordering can create (or eliminate) new cache conflict misses that the rescheduler will not recognize. Third, and we found this to be more significant than the first two, there is an indirect dependence between loads that access the same cache line. The first load to access the line essentially does a full or partial prefetch of the line for the other loads. If the loads are reordered, a different load sees the full latency of the access, and the original first load no longer does. These types of error also affect the slack measurements of [32].

Figure VI.1 illustrates tautness. Three fragments of a dataflow graph are shown, where the length of the boxes signify the latency of an instruction. In Figure VI.4(a), load A is a miss, and it fetches a line that is later used by load B. Since the original graph model described in section III.E does not include any edges to represent the cache line sharing dependence, it will appear, when the dependence graph is analyzed offline, that load A can be delayed without changing the critical path, which includes load B. That is, load A appears to have slack, as shown in Figure VI.4(b). However, if load A were in fact delayed in the processor, it would make load B a partial miss, as shown in Figure VI.4(c).

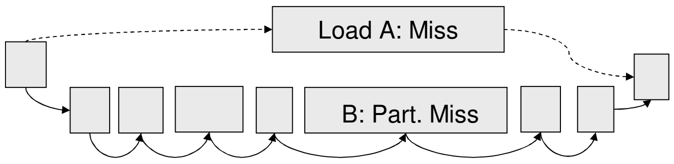
One solution is to add a dependence between the completion of A and load B. There are several ways to add this dependence, none of which are entirely satisfactory. One approach is to add a dependence between the E-node of a load that initiates a miss to a cache line, and the E-nodes of any subsequent loads



(a) Two loads which share a cache line. Load A is a miss, and load B is a hit, to the same line.



(b) Load A appears to have slack in the graph: it can be delayed without changing the critical path length.



(c) Actually, delaying load A turns B into a partial miss.

Figure VI.4: Example illustrating the effect of cache line sharing.

which access the same cache line, with a weight equal to the difference in the latency between the two loads. This insures that the first load will not appear to have undue slack. However, this does allow for the possibility that the second load could be the load which initiates the cache miss. In practice, this limitation may not be important. However, such a situation illustrates the difficulty in using a graphical model to represent all the details in a real processor, or even in a fairly detailed simulator.

VI.B.5 Unclassifiable Instructions

Although most dynamic instructions could be classified as critical or non-critical by our offline analysis technique, some were unclassifiable. An instruction may be unclassifiable for one of three reasons. Either it has a negative slack (increasing latency lengthens execution time by more than the increase in latency), it has a negative tautness (breaking dependences slows program execution), or because both slack and tautness are greater than zero (doing anything to an instruction seems to improve performance).

All of these unexpected behaviors come from the same source – the issue width constraint we have added to our graph model. Further efforts to more exactly model processor behavior would likely increase this phenomenon. But these are not incorrect or anomalous results – they represent real processor effects. Consider an otherwise non-critical instruction that becomes issuable in the same cycle as a critical instruction. If the critical instruction finds the issue window full, then an optimization that either speeds or delays the issue of the non-critical instruction accelerates the whole program by allowing the truly critical instruction to issue earlier. The unclassifiable instructions only occur when there is contention for issue bandwidth, and completely disappear if we remove this constraint from the rescheduler. Even negative values of slack and tautness are possible as a result of issue width constraints, although the magnitude of negative values is limited. It is fair to say that when a program is limited by issue width, the critical path is less well defined.

In summary, not all aspects of a processor can be captured by a purely graph-based model. However, it is still a very useful abstraction which can be used to predict with a useful level of accuracy, what instructions should be optimized or de-optimized.

VI.C Comparing Critical Path Predictors

Using the framework described in the previous section, we can know whether delaying each dynamic instruction is safe, and likewise, whether optimizing each dynamic instruction is worthwhile. We use this information about instructions to evaluate the usefulness of several proposed critical path predictors. We also explore the potential for new types of critical path predictors.

Critical path prediction differs from other types of prediction, such as branch prediction, in an important respect. In branch prediction, a predictor table is trained using the outcomes of each branch, and there is no ambiguity over what this training information is. In critical path prediction, there is also a predictor table, trained based on the criticality of each instruction, but identifying whether an instruction was critical is a large part of the challenge. Thus, the accuracy of a critical path prediction depends on both the accuracy of the training stream (identifying whether an instruction was critical after it executed) and the accuracy of predictions (how training information is used by the prediction table to predict future criticality). We study both the *identification accuracy* of different proposed methods, and the *prediction accuracy* when a perfect identification method is coupled with different prediction tables.

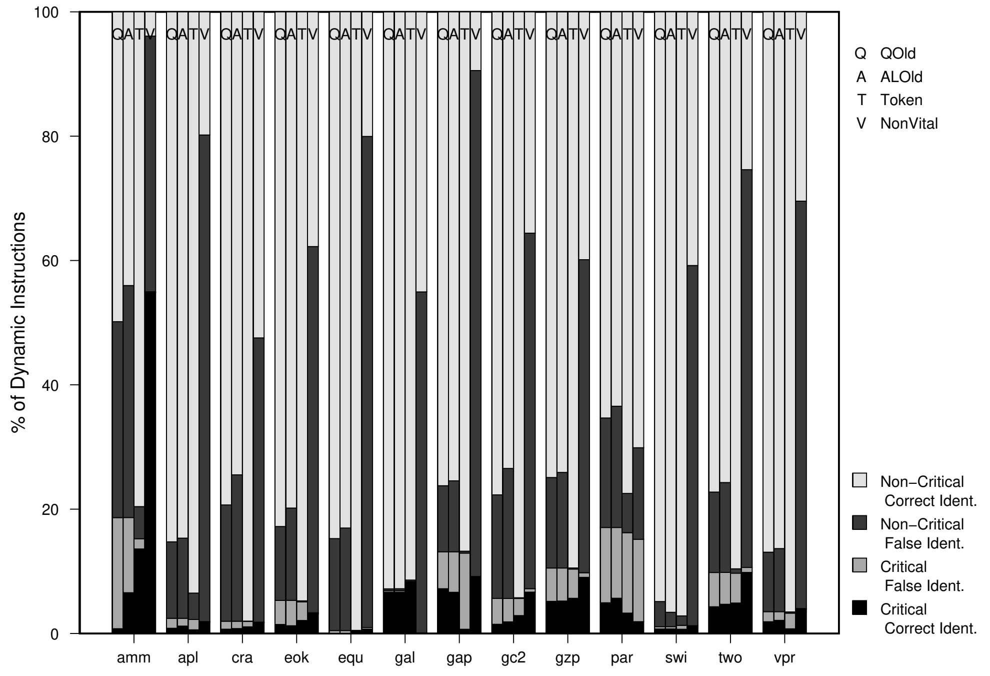


Figure VI.5: Correct and incorrect identification for 4 different criticality-identification mechanisms.

VI.C.1 Training Accuracy

This section examines the accuracy of the training mechanism used by several critical path predictors. We define an instruction as being non-critical if it has more than 0 cycles of slack, as measured by the rescheduler, or critical otherwise. Figure VI.5 shows the breakdown of correct and incorrect identification for various methods of identifying critical instructions across the 13 benchmarks. Each group of bars represents a benchmark. Within each group, individual bars represent different methods of identifying critical instructions. A letter at the top of each bar indicates the identification method. In this figure, *QOld* and *ALOld* represent critical path predictors from [117] using the QOld heuristic and the ALOld heuristic, respectively. QOld identifies instructions that become the oldest in the instruction queue, and ALOld identifies instructions that become the oldest in the active list (oldest non-retired). *Token* is the critical path predictor from [34], which plants a token at an instruction, and observes whether the token stays alive, propagating between instructions along last-arriving dependence edges. We modeled a token-passing predictor which could train 8 instructions at a time, with a 500-instruction training distance. *NonVital* is the load-criticality predictor from [87]. The NonVital predictor marks a load as "vital" if its result is used immediately.

The number of identifications made by each method varies. Thus, the ratio of the number instructions which are actually critical versus non-critical need not be the same for all bars in a group. The Heuristic methods (Q,A) make an identification for every instruction. The Token-passing predictor (T) only identifies a sample set of all instructions. The non-vital predictor only predicts loads. The exact fraction of identified instructions that are critical depends on the implementation of the predictor. For instance, in `ammp`, nearly half of all load instructions are critical, and in `galgel`, very few load instructions are critical, hence the fraction of critical instructions is clearly different for the NonVital bar as compared to the other predictors.

Figure VI.5 shows that the token-passing method does very well overall. It does particularly well at correctly identifying instructions that are actually non-critical – the heuristic techniques tend to be much more liberal in identifying potential critical instructions. For many optimizations, however, the most important category can be the mis-identification of critical instructions. For example, in a processor with clustered functional units, mistakenly sending a critical instruction to a different cluster from other critical instructions will have a direct cost: increased execution time due to bypass penalty. But mistakenly sending a non-critical instruction to the wrong cluster only has an indirect cost: possibly causing contention. The token-passer also does well with this criteria overall, but in several cases does not have the highest coverage of critical instructions.

The Non-Vital predictor (V) was proposed only as a predictor for Load instructions. The results show that the Non-Vital Loads technique does especially well at identifying critical loads, but it does poorly at correctly identifying non-critical loads.

Note that the Token-passing predictor rarely identifies non-critical instructions as critical, because its prediction is based on a graph the same graph we use to measure criticality. But because we define criticality based on the slack measured using the rescheduler, which incorporates an additional processor constraint (limited issue width), such mispredictions are possible.

It may seem that the Token-passing method and the non-vital method would never fail to identify a critical instruction–that is, all their errors would be mis-identification of a non-critical instruction. Both the token-passing method and the non-vital method would identify an instruction as non-critical when its result is not used immediately. However, the definition of a non-critical instruction, for the purpose of this analysis, is that an instruction can be delayed some number of cycles greater than 0 without delaying the program. Although some instruction *i* may not have its result used immediately by any dependent, it may still be critical because delaying *i* causes some a dependent instruction to experience extra delays due to a functional unit-contention that it would not have otherwise experienced.

VI.C.2 Criticality Bias of Static Instructions

Given a means to accurately identify which instructions are critical, the critical path predictor then uses that information to produce a prediction for future instructions.

Benchmark	>99%	>95%	>90%	>50%	>10%	>1%	>0%
amm	3.38	3.64	3.64	6.59	13.69	18.63	20.36
apl	6.66	6.66	6.67	7.39	15.83	19.30	22.49
cra	0.67	0.72	0.78	2.20	11.86	25.70	34.68
eok	0.45	0.51	0.58	1.15	6.58	12.03	18.02
equ	0.00	0.00	0.00	0.00	0.85	2.55	3.77
gal	0.00	2.18	4.37	4.37	4.80	16.16	38.86
gap	0.22	0.48	0.63	2.82	14.48	23.35	30.87
gc2	2.59	2.62	2.65	3.79	15.61	29.96	34.49
gzp	3.34	3.43	4.34	6.90	21.18	27.13	38.59
par	3.29	3.46	3.70	7.27	20.95	32.56	37.15
swi	0.00	0.00	0.00	0.00	11.60	19.89	25.97
two	0.40	0.40	1.03	4.53	20.79	35.31	51.93
vpr	0.14	0.14	0.14	0.55	6.57	10.82	17.36
AVG	1.48	1.71	2.07	3.41	12.59	21.23	29.52

Table VI.3: The criticality bias of static instructions.

If criticality is highly biased for individual static instructions, a simple predictor, even a static predictor, would be sufficient. This section examines the criticality bias of static instructions for a particular processor configuration. The results are shown in Table VI.3. In that table, a column labeled $x\%$ shows, for each benchmark, the fraction of static instructions for which more than $x\%$ of its dynamic instances were critical.

Looking at the last column of the table, between 4% and 52%, and on average 30% of static instructions are critical at least once. Thus, on average, 70% of static instructions can safely be ruled out as not being critical. This suggests that techniques that need to find a large number of, but not necessarily all, instructions with slack may not require a complex predictor.

Among static instructions which are at any point in time critical, those instructions tend to change their criticality often. For three of the benchmarks (equake, swim, and vpr), less than 1% of static instructions are critical even half of the times they are executed. That is, there are virtually no "statically critical" instructions for those benchmarks.

If we want to try to predict the critical path of the program with high

Benchmark	$P > 0.9$	$P > 0.5$	$P > 0.1$	$P > 0.01$
amm	0.35	1.91	12.74	18.20
apl	7.73	8.83	17.05	19.61
cra	0.53	1.50	15.05	28.53
eok	0.10	0.79	7.44	13.32
equ	0.00	0.00	1.60	2.55
gal	0.00	0.00	0.44	19.21
gap	0.78	2.88	16.65	24.93
gc2	2.48	3.78	18.80	32.03
gzp	4.30	4.90	22.27	29.08
par	2.82	5.00	23.82	34.03
swi	0.00	0.00	14.92	21.55
two	0.00	1.60	26.25	39.21
vpr	0.08	1.40	8.00	11.77
AVG	1.57	2.56	14.36	22.99

Table VI.4: The fraction of static instructions which change criticality with different frequencies, by frequency.

accuracy, then a purely PC based approach will not be sufficient. According to the table, on average over all benchmarks, 33% of static instructions are critical at least once (column labeled 0%) and 12% of static instructions are critical more than 50% of the time. Thus, 21% are critical, but at a frequency less than or equal to 50%.

However, this table does not indicate whether static instructions are changing their criticality only occasionally, such as when the program enters different phases of behavior, or whether they are changing their behavior rapidly. Table VI.4 shows the distribution of static instructions according to how often they change criticality (from critical on one dynamic instance to non-critical the next, or vice versa.) In this table, a column labeled $P > x$, with value y for some benchmark, means that y% of static instructions have a probability greater than x of changing their criticality between any two subsequent instances. For example, a static instruction that was critical 50 times in a row, and then non-critical 50 times in a row, and so on, would have $P = 2\%$. A different static instruction that alternates between critical and non-critical every time would have $P = 100\%$.

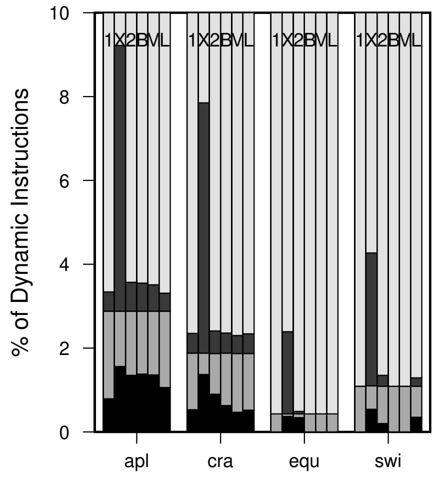
This table shows that on average 23% of static instructions tend to change their criticality more often than every 100 instances. Thus, a predictor which identifies the criticality of a static instruction, say, every 10 instances, would be able to predict 75% of static instructions with reasonable accuracy. This bodes fairly well for predictors like the token-passing predictor that produce intermittent training information. However, predictors that produce more frequent training information, like the Heuristic predictor, may still be able to use that to an advantage. 14% of static instructions overall (and up to about 25% for some benchmarks) change criticality at every 10 invocations or more. For example, consider a load instruction that has a cache miss every 8th time, because it reads 8 sequential words from a cache line. That instruction or its dependents might be non-critical 7 times and then critical 1 time, and repeating in that pattern. Predicting criticality based on a saturating counter would not be effective for this instruction. The next section examines the possibility of identifying patterns and correlations which can increase the predictability of the dynamically critical instructions.

In order to correctly predict the criticality of such a frequently changing instruction, it would be necessary to use a predictor which uses more than just the PC of the instruction to make a prediction, but which uses additional history or correlating information to predict the criticality of an instruction. We perform an evaluation of such predictors in the next section.

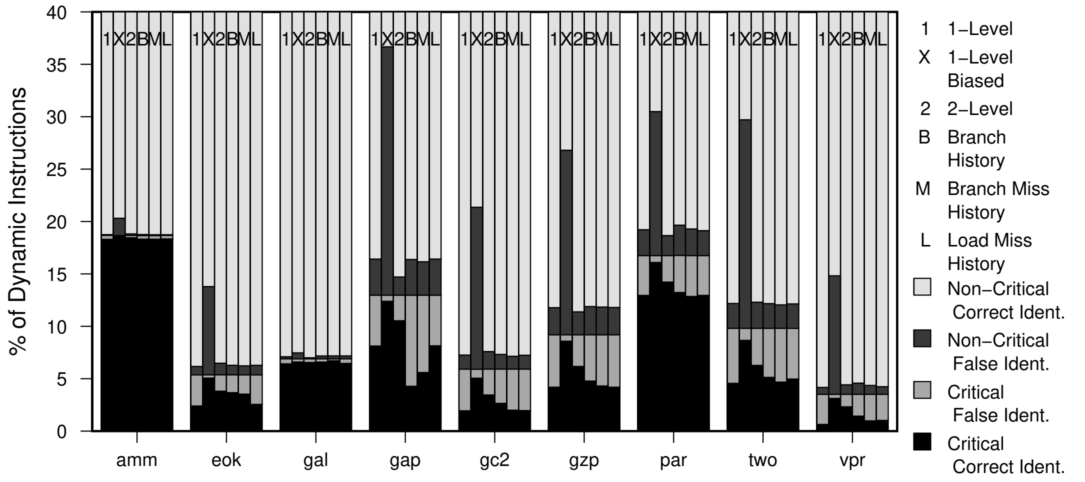
VI.C.3 Prediction

Previous work in critical path prediction used a PC-indexed prediction table with biased counters[2]. This means that the address of a an instruction is used to index into a table of saturating counters, like a branch predictor, and that the counters are incremented by a large amount when an instruction is

[2]Srinivasan, [105], used some global branch history to index their load-criticality predictor.



(a)



(b)

Figure VI.6: The number of correct and incorrect predictions for different prediction tables, all using oracle criticality identification.

identified as critical, and decremented by 1 when an instruction is found to be non-critical. Both the token-passing predictor and the heuristic predictor used such a prediction table. We examine the accuracy that can be obtained with different types of prediction tables. In this section, we are less concerned with the practicality of implementing a prediction table, and more interested in the limits of how well critical instructions can be predicted. Therefore, we use Oracle training for all the predictors in this section.

Figures VI.6(a) and VI.6(b) show the accuracy of different types of prediction tables. Each group of bars represents a benchmark. Within each group, individual bars represent different predictors. A code at the top of each bar indicates the predictor. The predictors are as follows: **One-Level** (**1** in the figure) – A one-level predictor consisting of a PC indexed table of 2-bit saturating counters. The table is un-aliased. **1-Level, Biased Counter (X)** – A one-level predictor consisting of a PC indexed table of 5-bit saturating counters. The biased counter increments by 8 when an instruction is identified as critical, and decrements by 1 otherwise. **Two-Level (2)** – A two-level predictor, PC indexed table of 8-bit local histories (un-aliased). Local history is used to index into a table of 256 2-bit saturating counters. **Branch History (B)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of branch-direction history, un-aliased. **Branch Miss History (M)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of branch-misprediction history, un-aliased. **Load Miss History (L)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of load-miss history, un-aliased. All predictors use oracle training. Note that there are two sub-plots with different vertical scales, to show detail on those benchmarks that have a small percentage of critical instructions.

Each of the history-based predictors (BranchHistory, BranchMissHistory, LoadMissHistory) use 8 bits of history regarding the last 8 branch or load instructions. BranchHistory refers to the direction of previous branches. BranchMissHistory refers to whether previous branches were mispredicted. LoadMissHistory refers to whether previous loads were cache misses. Previous means older instructions, in program order. Information about the current instruction is not incorporated in the history.

Branch history would help prediction if the criticality of an instruction is highly correlated with the control flow path the program took to get to it. Branch history (B) is sometimes better than a simple PC-indexed prediction (1), and sometimes worse. The criticality of instructions can be affected by nearby cache misses and branch mispredictions, but these patterns did little to improve prediction accuracy except in the isolated case of `swim`. Note that miss history and mispredict history are hard to gather in real time, but our focus was on understanding the causes of varying criticality.

Branch history is information that might reasonably be used to improve critical path prediction, since it is already used in conjunction with branch prediction. We thought that branch mispredictions and cache misses would be two events that could cause the criticality of subsequent instructions to change. Branch mispredictions and load misses are not normally known ahead of time, thus it is not likely to be practical to use such histories to improve critical path prediction. However, our interest is in understanding why the criticality of instructions varies, and the limits of how well criticality can be predicted.

These results reinforce the findings in the previous subsection; Most static instructions are always not-critical, and so all of the predictors are able to identify nearly all non-critical instructions. The critical path runs through a small set of static instructions, but which of those are critical can vary frequently. Thus, if we are willing to tolerate predicting some non-critical instructions as critical (accept low accuracy to get high coverage of critical instructions), then the best approach is to predict as critical any static instruction that was recently critical. This is highlighted by the "Critical/Correctly Identified" bar for "Perfect-OneLev-Biased", which is always very tall. Thus the approach taken by two critical path predictors [117, 34] of biasing the counters (by incrementing by a large amount for critical, and decrementing by a small amount for non-critical) is effective.

However, if we are not willing to sacrifice accuracy, the two-level predictor had significantly higher coverage of critical instructions among the those that were not biased (few non-critical instructions called critical). In several cases, it achieves twice the coverage of the PC-based 1-level predictor. This indicates that many instructions do indeed follow a predictable pattern of criticality that can be identified by a local history predictor.

However, the two-level predictor would not be compatible with a sampling-based identification method, such as the token-passing predictor. Recall that the token-passing method does not produce training information for every instruction, but has a better training accuracy than the heuristic methods, which do sample every instruction. Although tokens can be placed in a controlled fashion to profile several consecutive instances of a static instruction, this would need to be continued indefinitely to maintain the local history for an instruction.

This section has demonstrated that predictors that predict based on PC and are slow to change predictions have a difficult time predicting critical instructions accurately, due to the highly dynamic behavior of those instructions. It has shown the potential for a pattern-based predictor to be more effective. The patterns by which instructions change their criticality warrants further study. One likely source of predictable patterns of criticality could stem from loads that access memory sequentially, missing for the first access to a line, and then hitting on subsequent accesses to that line.

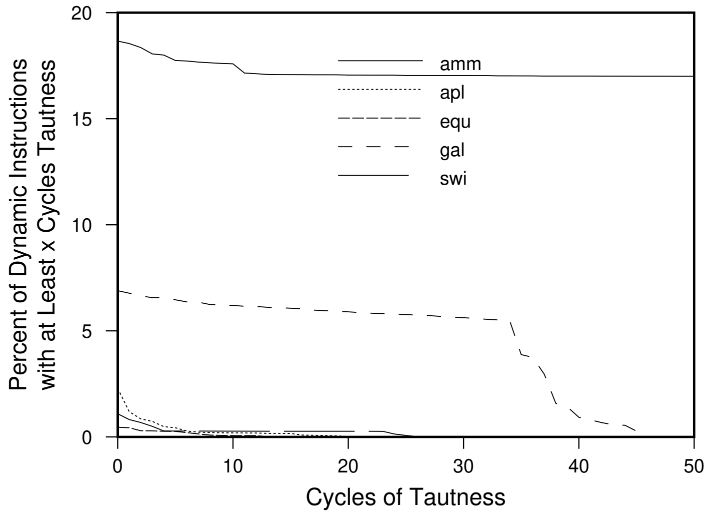
VI.D Distribution of Critical Instructions

Slack and tautness are two metrics that provide more fine-grained information about criticality than a binary prediction (critical vs. non-critical). This section confirms this by showing slack and tautness are both highly varied in the set of programs we are considering.

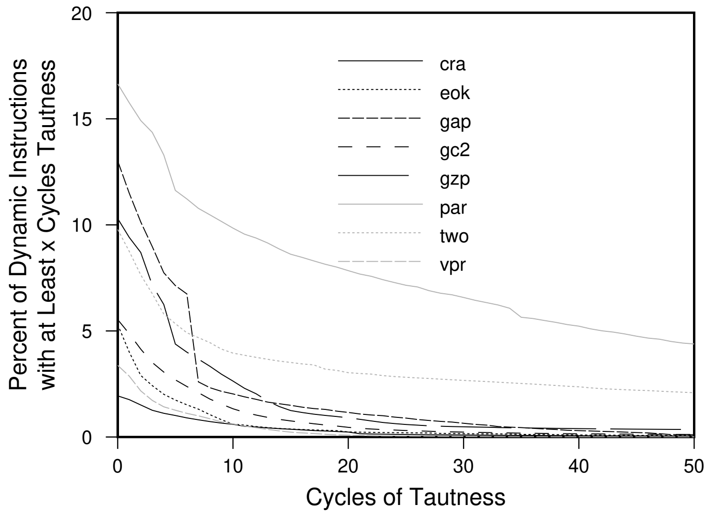
Figures VI.7(a) and VI.7(b) show the cumulative distribution of tautness values for dynamic instructions in the floating-point and integer benchmarks. A point on the curve shows what percentage of instructions have at least a certain number of cycles of tautness. Where a curve intersects the y-axis indicates the percentage of dynamic instructions that are critical.

Most integer benchmarks (the graph on the right) have fewer than 2.5% of instructions with tautness of more than 10 cycles. The benchmarks twolf and parser stand out as exceptions. Three of the 5 floating-point benchmarks have fewer than 1% of instructions with tautness of more than 10 cycles. We attribute the reduced amount of tautness in some floating point programs in part to loop unrolling and instruction scheduling, which will increase the number of similar-length, independent dependence chains, so that optimizing just one instruction (as tautness measures) will leave several other equally long, parallel chains which become critical. Since we define tautness in terms of making the result of an instruction available as soon as it is dispatched, rather than just reducing its latency, it is possible for an instruction to have a tautness much greater than the longest latency of any instruction in our simulator (about 360 cycles). This is most evident in `ammp`, which has a significant number of instructions with a tautness greater than 2000 cycles. Several of the integer programs also have a significant number of instructions with hundreds of cycles of tautness, as indicated by the long tails on the curves.

Figures VI.8(a) and VI.8(b) show histograms of the amount of slack in

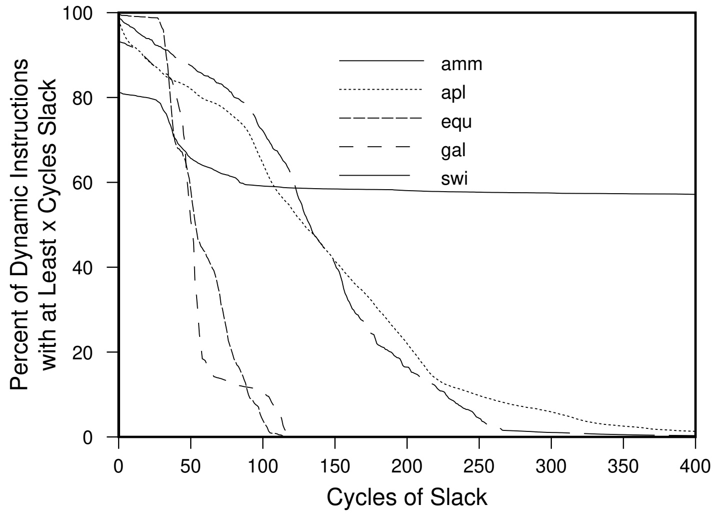


(a) Benchmark group 1

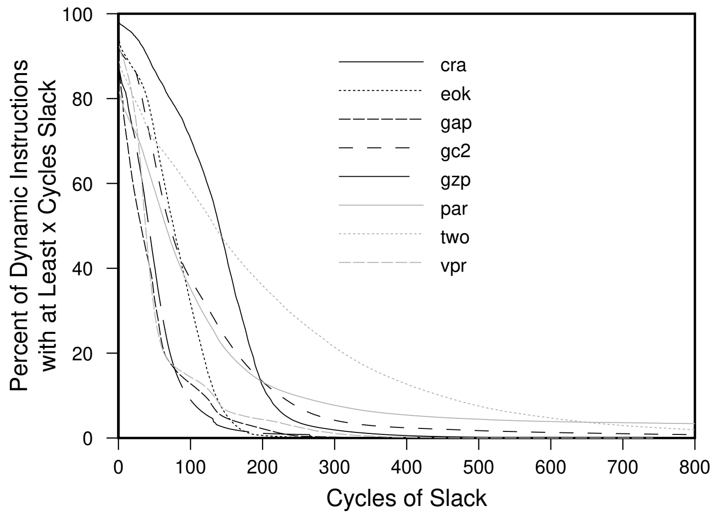


(b) Benchmark group 2

Figure VI.7: Cumulative distribution (decreasing) of the fraction of dynamic instructions with certain tautness.



(a) Benchmark group 1



(b) Benchmark group 2

Figure VI.8: Cumulative distribution (decreasing) of the fraction of dynamic instructions with certain slack.

Name of Policy	Select the top n static load instructions according to:
Times-Critical	Number of dynamic instances of a static load which were critical, according to the rescheduler.
Tautness	Sum of tautness, as measured by the rescheduler, for each dynamic instance of a static load.
Available Tautness	Sum of available-tautness for each dynamic instance of a static load.
L1-Misses	Number of L1 cache misses for a static load.
L2-Misses	Number of L2 cache misses for a static load.
Latency	Sum of latencies of each dynamic instance of a static load.

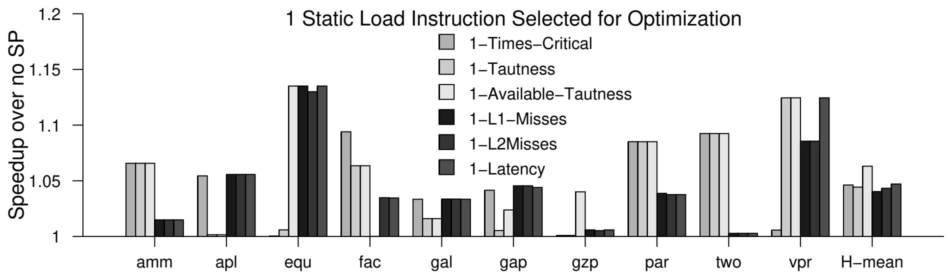
Table VI.5: Policies used for selecting static loads for optimization.

dynamic instructions. Notice that, particularly for the integer programs, there is a correlation between programs with high tautness values and high slack values. The average number of instructions with slack is much higher than the number of critical instructions, which was one of the original motivations for critical path prediction.

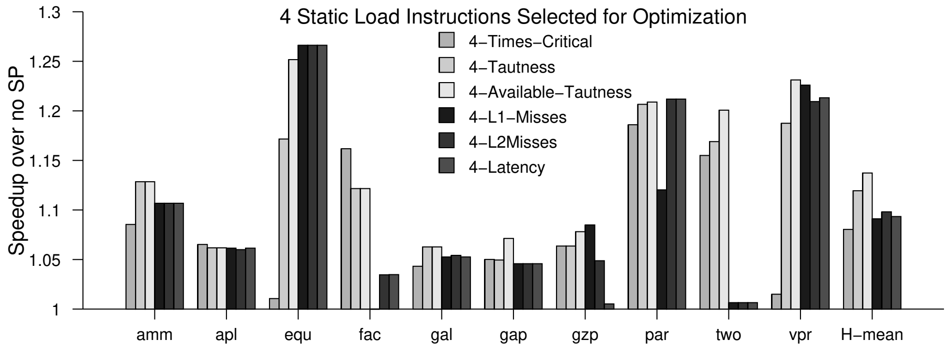
VI.E Applying Criticality Information

This chapter introduces the idea that we can quantify the criticality (tautness) of individual instructions. If this metric is useful, it has two implications for critical path predictors. First, if a predictor does not predict all critical instructions correctly, it is more important that it correctly predicts those with higher tautness. Second, a predictor that accurately produces a tautness prediction rather than a criticality (binary) prediction should be more effective. This section verifies the second implication.

We apply a (perfectly trained) tautness prediction to guide a hypothetical optimization which reduces the latency of load instructions. This hypothetical optimization is intended to emulate speculative precomputation. Speculative precomputation [23, 22] is an optimization where a speculative hardware thread is



(a) 1 static load chosen.



(b) 4 static loads chosen.

Figure VI.9: Idealized speculative precomputation with different ways of choosing static load instructions to optimize, with 1 or 4 static loads chosen.

executed in parallel with the main program. The speculative thread consists of a subset of the instructions that the main thread will execute. The speculative thread initiates long latency memory operations earlier, thus allowing the main program to run faster. Speculative precomputation makes an interesting test case because the cost of each targeted load is high (requires a hardware thread context on a multithreaded processor), thus the total number of targeted loads should be kept low. Collins et. al. [23] selected loads for precomputation based on the number of L1 caches misses. In the subsequent work, [22], loads were selected using a method derived from a heuristic criticality predictor from [117].

We measure the speedup on an idealized form of speculative precomputation when different methods are used to select static load instructions. The results are presented in Figure VI.9. In our idealized optimization, there is a limit to the number of static load instructions that can be selected for optimization (1 or 4 static loads, for the results shown here). When a static load is “optimized”, all dynamic instances of that load hit in L1 cache. The line which it accesses, if not already in cache, always arrives just in time for the optimized load to read it when it executes. Non-optimized loads which access data from the same line may benefit as a side effect.

We consider 6 different ranking criteria, to choose the best n static loads for optimization. Loads are ranked according to the criteria and the top n are selected. Three ranking criteria are based on criticality, and 3 are based on latency. Table VI.5 summarizes the ranking criteria for static loads.

The *L1-Misses* ranking criteria counts the number of L1 misses that a static load experiences. [23] choose L1 misses as a criteria since they were studying SP on an in-order processor. An out-of-order processor may be better able to tolerate L1 cache misses, hence we also include the *L2-Misses* ranking criteria. We also choose loads using the *Latency* criteria, which picks static loads for which the sum of the latency of the dynamic instances is greatest.

The *Times-Critical* criteria ranks a load by the number of times it was critical during a segment of a program. The *Tautness* metric ranks a load by the sum of the tautness of each instance of the load, as measured by the rescheduler. Recall that tautness represents the maximum reduction in cycles that could be achieved by optimizing a particular dynamic instruction. Ideally, the sum of the tautness would correspond to the number of cycles by which the execution time would be reduced. However, optimizing just one dynamic instruction affects the critical path later on, and thus the tautness of later instructions would be changed. So while we would not expect to enjoy the total execution time reduction suggested by the sum of tautness for all dynamic instances of an instruction, we do expect that it should be indicative of the potential for speedup.

The tautness metric measures the amount of speedup possible if the data dependences originating at a particular instruction are removed entirely, and thus the most improvement that a single instruction optimization might realize. But tautness is only a maximum. For an optimization like SP, where the latency of a load is reduced though prefetching, but no dependences are broken, the reduction in cycle time that can be realized may be less than the tautness. We define the *available tautness* of an instruction with respect to a latency-reducing optimization to be $\min(t, l_0 - l)$, where t is the tautness of that instruction, l is its latency after being optimized, and l_0 is its unoptimized latency. Thus, if an instruction has a tautness of, say, 10 cycles, but you only reduce its latency by 4 cycles, then we would expect that the execution time would only be reduced by 4 cycles. Thus, *Available-Tautness* chooses the static loads which have the greatest available tautness, summed over all dynamic instances.

Figure VI.9 shows the performance when different policies are used to choose static load instructions. All are presented as speedup over execution with no SP applied. For each of these metrics, loads were selected using profiling data from a 10 million instruction program segment, and speedup measurements were taken over a 100 million instruction program segment. This is representative of how, in a dynamic implementation of SP, the loads would be profiled, threads built, and then those threads would be used for an extended period of time.

While no criteria always chooses the best set of loads to optimize, the harmonic mean of the speedup is greatest with the *Available-Tautness* metric for both 1 and 4 loads. The *Times-Critical* metric, on the other hand, performs poorly. The difference between the *Available-Tautness* metric for 4 loads, and the best latency-based metric, *L2-Misses*, is 4% speedup. In the best case, the speedup of twolf is increased from a negligible 0.6% speedup under all three latency-based criteria, to 20% speedup, using *Available-Tautness*. Over the program segment profiled, twolf has 820 static load instructions, many of which experience many L1 and L2 cache misses. However, the most frequently executed of those loads are well scheduled, and the approaches which are ignorant of the critical path do not choose appropriate loads.

The effectiveness ratio from using tautness information (*Available-Tautness*) versus the *L2-Misses* criteria is 46% and 40%, for 1 and 4 static loads. (Effectiveness ratio was used to evaluate critical path predictors in [117] and [34], and quantifies the extent to which CP prediction makes an optimization *more* effective.)

This experiment demonstrates the existence of optimizations for which a binary prediction of criticality is not as effective as a prediction that also produces an estimate of the benefit of optimizing that instruction.

VII

Balanced Multithreading

In this chapter, we consider the relationship between critical-path aware optimization and two important trends in microarchitecture: multithreading, and the processor–memory gap. These two trends complement each other: the growing processor–memory gap reinforces the need for multithreading. We do not attempt to comprehensively consider the interaction between multithreading and critical-path prediction in this thesis. We do evaluate a new multithreading architecture which is inspired by our studies of criticality, and which is guided by a per-thread critical-path summary metric.

Critical path prediction helps make better use of a processors microarchitectural resources by identifying which instructions must be optimized to improve performance, and which instructions can tolerate lesser treatment. As the execution time of critical path instructions is shortened, and as non-critical instructions are delayed, the number of parallel operations will increase, within the limitations imposed by hardware on parallel execution.

In practice, only so much can be done to optimize critical instructions. For some time, a gap has been growing between processor clock periods and main memory access times. Although a wealth of techniques exist to reduce the number and effect of cache misses, some long latency accesses will still remain.

This leaves the processor hardware underutilized by a single program.

Multithreading is an architectural technique by which the processor may fetch and execute instructions from several programs or threads without operating system intervention. Multithreading is especially useful in the face of this processor–memory gap, because, when one thread is waiting for memory, the remaining can continue to make good use of the processor. Many recent and announced processor designs support some form of multithreading [54, 12, 85, 49, 27]. We provide some background on multithreading in Section II.C.

In one sense, critical-path prediction and multithreading are complementary. Multithreading increases the number of concurrent operations in a processor, which in turn increase contention for resources, and may reduce the fraction of instructions which lie on the critical path. Critical-path prediction provides a guide to arbitrate among instructions for these contended resources. There is one complication here: in a multithreaded processor running several independent programs, performance should be measured by *throughput*: the rate at jobs are completed, from a continuous stream of independent, unit sized jobs. Critical-path analysis does not provide any guidance on prioritizing completely independent tasks. We suggest that critical-path prediction can still be used, with a small caveat. Critical-path prediction can be used to identify which instructions are critical to the progress of an individual thread, without regard to other threads. When the processor must arbitrate between a critical and a non-critical instruction, the critical instruction should get priority. The application of the critical-path predictors described in this thesis to a multithreaded processor is the subject of an ongoing research effort.

The relation between critical-path prediction and the growing processor memory gap is less complementary. Basic arithmetic operations, like integer addition, require a single cycle to execute on a contemporary processor, while a load instruction which accesses memory can require several hundred cycles. Many programs do not have chains of several hundred operations which are independent of a load miss. Critical-path prediction may not be needed to determine whether a long-latency load miss is critical. It is safe to guess that most long-latency load misses are critical. If they are not, it is most likely because a second long-latency load miss occurs in parallel with the first. The new multithreading architecture which we propose in this chapter makes use of these two observations, and thus it does not directly make use of critical-path prediction.

Although our multithreading proposal stands on its own, it is related to and inspired by the work described in the previous chapters. As discussed in Sections VI.B and VI.D, many instructions have high tautness values. As defined previously, tautness indicates the maximum benefit from speeding-up the execution of some instruction. This definition can be viewed another way. If instruction i has a tautness t, then the instructions which follow i either depend on i, or have t cycles of slack. Thus, when a critical instruction i, with tautness t, is executing, then we could not execute any other instructions for t cycles, and still not effect the performance of the program. The longest latency instruction, and thus those with the highest effective tautness, are loads. In other words, there is often not enough instruction level parallelism to hide the cost of long latency load instructions. This suggests an architectural optimization: when a high-tautness load instruction is executing, we can move out all the other instructions from the pipeline. In fact, the loads memory request can continue even if the load instruction is flushed from the pipe (to be refetched later). With all instructions out of the pipeline, the whole pipeline is free to execute another program. This is not a new idea; it is called coarse-grained multithreading. However, the rest of this Chapter explores a novel approach to coarse-grained multithreading, and its combination with simultaneous multithreading.

VII.A Simple Multithreading

The ratio between main memory access time and core clock rates continues to grow. As a result, a processor pipeline may be idle during much of a programs execution. A multithreading processor can maintain a high throughput despite a large relative memory latencies by executing instructions from several programs. Many models of multithreading have been proposed. They can be categorized by how close together in time instructions from different threads may be executed, which affects how the state for different threads must be managed. Simultaneous Multithreading [115, 112, 50, 126] (SMT) is the least restrictive model, in that instructions from multiple threads can execute in the same cycle. This flexibility allows an SMT processor to hide stalls in one thread by executing instructions from other threads. However, the flexibility of SMT comes at a cost. The register file and rename tables must be enlarged to accommodate the architectural registers of the additional threads. This in turn can increase the clock cycle time and/or the depth of the pipeline.

Coarse-grained multithreading (CGMT) [3, 90, 108] is a more restrictive model where the processor can only execute instructions from one thread at a time, but where it can switch to a new thread after a short delay. This makes CGMT suited for hiding longer delays. Soon, general-purpose microprocessors will be experiencing delays to main memory of 500 or more cycles. This means that a context switch in response to a memory access can take tens of cycles and still provide a considerable performance benefit. Previous CGMT designs relied on a larger register file to allow fast context switches, which would likely slow down current pipeline designs and interfere with register renaming. Instead, we describe a new implementation of CGMT which does not affect the size or design of the register file or renaming table.

We find that CGMT alone, triggered only by main-memory accesses, provides unimpressive increases in performance because it cannot hide the effect of shorter stalls in a single thread. However, CGMT and SMT complement each other very well. A design which combines both types of multithreading provides a balance between support for hiding long and short stalls, and a balance between high throughput and high single-thread performance. We call this combination of techniques *Balanced Multithreading* (BMT).

This combination of multithreading models can be compared to a cache hierarchy, which results in a *multithreading hierarchy*. The lowest level of multithreading (SMT) is small (fewer contexts), fast, expensive, and closely tied to the processor cycle time. The next level of multithreading (CGMT) is slower, potentially larger (fewer limits to the number of contexts that can be supported), cheaper, and has no impact on processor cycle time or pipeline depth.

In our design, the operating system sees more *virtual contexts* than are supported in the core pipeline. These virtual contexts are controlled by a mechanism to quickly switch between threads on long latency load misses. The method we propose for adding more virtual contexts does not increase the size of the physical register file or of the renaming tables. Instead, inactive contexts reside in a separate memory dedicated to that purpose, which can be simpler and far from the core as compared to a register file, and will not be timing critical. Further, those threads that are swapped out of the processor core do not need to be renamed, which avoids an increase in the size of the renaming table. This architecture can achieve the throughput near that of a many-context SMT processor, but with the pipeline and clock rate of an SMT implementation that supports fewer threads. We find that we can increase the throughput of an SMT processor design by as much as 26% by applying these small changes to the processor core.

Some background on multithreading and a discussion of related work is presented in Section II.C. The rest of this chapter is organized as follows: Section VII.B presents the architecture and mechanisms for combining SMT and CGMT. Section VII.C discusses our evaluation methodology. Results are presented in Section VII.D.

VII.B A Balanced Multithreading Architecture

Implementations of SMT processors thus far have been conservative; the Pentium 4 processor [75] and each core of the Power 5 processor [54] have two hardware contexts. Although the potential exists for much higher throughputs when more than 2 contexts are presented to the operating system, just adding more physical contexts to SMT alone may not be the best way to achieve higher throughput. Each additional context supported by an SMT processor increases the size of the register file and of the renaming table. This in turn increases the pipeline length or clock cycle time, which penalizes performance, especially when only one or a few threads are available to run. Although at least one SMT processor design with 4 contexts has been attempted, [29, 30, 77], it never reached production. Redstone, in [88], citing [85], points out that the register file of that processor, the Alpha 21464, would have been 3-4 times the size of its 64k data cache. Likewise, previous work on coarse-grained multithreading [3, 12] used a larger register file to hold the register values of an inactive thread. Instead, we propose extending an SMT processor design by adding a coarse-grained multithreading mechanism which requires few hardware changes.

The Sparcle CPU [3] in the Alewife machine implements CGMT. It performs a context switch in 14 cycles (4 cycles with aggressive optimizations). The Sparcle architects disabled the register windows present in the Sparc processor that they reused, and used the extra registers to support a second context. The Sparcle processor was in-order, with a short pipeline and did not perform register renaming. The IBM RS64 IV processor [12] supports CGMT with 2 threads, and is in-order. The RS64 designers chose to implement only two contexts, which avoided any cycle-time penalty from the additional registers. For a processor with a large instruction windows backed by additional registers, the register file access time is much more likely to be on the critical timing path.

Waldspurger and Wiehl [120] avoid expanding the register file in a CGMT architecture by recompiling code so that each thread used fewer registers. Mowry and Ramkissoon [78] propose software-controlled CGMT to help tolerate the latency of shared data in a shared-memory multiprocessor. They suggest compiler-based register file partitioning to reduce context-switch overhead. Horowitz et al. similarly suggest using memory references which cause cache misses to branch or trap to a user-level handler [51]. Our approach uses lightweight hardware support to make context switches faster than would be possible purely using software, and does not require recompilation.

To reduce the incremental cost of additional threads in an SMT processor, Redstone et al. [88] propose partitioning the architectural register file. Lo et al. [72] propose software-directed register deallocation to decrease dynamic register file demand for SMT processors. Both [88] and [72] require compiler support. Multi-level register file organizations reduce the average register access time [10, 26, 8].

Register file speed is a function of the number of ports, as well as the number of registers it contains. A processor with a high issue width requires a register file with many ports to avoid contention. The port requirements can be relaxed [80, 62, 110], but that requires additional hardware to arbitrate among the ports.

VII.B.1 Terminology

In this chapter, we use the term *context* to refer to the hardware which gives a processor the ability to run a process without operating system or software intervention. We use the term *thread* to refer to a program assigned to a context by the operating system. Because the BMT architecture we propose exposes more contexts to the operating system than can be active at once in the processor core, we distinguish between *physical contexts* and *virtual contexts*.

The number of physical contexts, denoted C_{phys}, is the number of threads which can have instructions in the pipeline simultaneously, and is limited by the register file and renaming table sizes. The number of *virtual contexts*, denoted C_{virt}, is the total number of threads which are supported at once, via CGMT. For an SMT-only processor, $C_{virt} = C_{phys}$. We refer to an SMT-only processor design as being an SMT-C processor design when it has C contexts. For example, the Pentium 4 is an SMT-2 processor. We refer to a Balanced Multithreading design with C_{phys} physical contexts and C_{virt} virtual contexts as a BMT-C_{phys}/C_{virt} processor. If the Pentium 4 were extended using our techniques to present 2 more contexts to the operating system, for a total of 4 virtual contexts, then we term it a BMT-2/4 processor. Because there are more virtual contexts than physical contexts in a BMT processor, some threads will be *inactive* at any given time. An *inactive thread* can have a pending main memory request, but, unlike an *active thread*, an inactive thread does not have instructions in the pipeline nor does it have values in the primary register file.

VII.B.2 Firmware Context Switching

We propose a context switching mechanism which (1) does not increase the size of the register file because architectural state of inactive threads is stored elsewhere, (2) does not increase the number of ports on the register file, because the save/restore instructions access the register file like ordinary instructions, (3) does not affect the design of the renaming table, because inactive threads have no instructions in the pipeline, and (4) is considerably faster than a software context switch by the operating system. This mechanism, which we call *firmware context switching*, uses:

1. an exception-like mechanism to initiate a context switch and to flush the pipeline,
2. a microcoded instruction sequence of special instructions to swap the register state of active and inactive threads.
3. a separate buffer to hold architectural registers of inactive threads,
4. a small amount of duplicated or additional hardware in areas that should not be critical to performance.

We now describe the features of firmware context switching in greater detail.

Detecting Load Misses and Flushing—When a load instruction needs to directly access main memory, a thread swap may be initiated. A firmware context switch is not fast enough to make thread switching profitable for loads which hit in a second or third level cache, given current cache latencies. We use a simple method to detect main memory accesses: if a load has an execution latency over a certain threshold, the load is assumed to be accessing main memory. When such a load is detected, it is *canceled*, but its memory request remains in the memory system. In the commit stage, the load instruction will raise an exception when it is the oldest instruction in its thread. Fetching from that thread stops, instructions from that thread are flushed, the PC of the cancelled instruction is saved, and the register map is restored to point to the proper state. However, instead of jumping to a trap handler, control is transferred to a microprogrammed instruction sequence.

One side effect of canceling an instruction, as we do with long latency loads, is that the possibility of livelock is introduced. Kubiatowicz gives a thorough treatment of these issues in [64]. To avoid livelock in our simulations, we require that a thread commit at least one instruction before it can be swapped out.

Tullsen and Brown [111] note that very long latency memory operations can create problems for an SMT processor. They suggest that when a thread is stalled waiting for a memory access, the instructions after the miss should be flushed from the pipeline, freeing critical shared execution resources. Our scheme inherently provides the same functionality. However, their proposal fails to free the most critical shared resource – thread contexts. We compare our processor designs against an SMT processor which implements their flushing mechanism. Our results show that freeing resources being held by a stalled thread is indeed very important; however, making those same resources available to a thread that would not otherwise have a chance to run is also important. Other researchers have suggested more sophisticated flushing policies for SMT [21], which we do not evaluate. However, improvements to policies which control when to flush an SMT processor can also be applied to controlling thread-swapping in a BMT processor.

Microprogrammed Context Switch—After a thread has been flushed, instructions are fetched from a microcode control store. This microprogram consists of (1) a sequence of store-like `rsave` instructions, (2) a special thread-switch instruction, and (3) a sequence of load-like `rrestore` instructions. Each of the `rsave` and `rrestore` instructions is renamed, issued, and executed on an integer unit like a normal instruction. They are like a load or store instruction in they have one register operand, but they do not access programmer-visible memory space or undergo address translation. Instead they access a special buffer, the *Inactive Register Buffer* (IRB), which is described below. The address in the IRB is implicit given the operand and thread associated with an `rsave`/`rrestore` instruction. An unoptimized microprogram would have one `rsave` and one `rrestore` instruction for each architectural register.

We add two optimizations to this microcode sequence which reduce the number of instructions in a context switch. First, a *Dirty Register Mask* (DRM) tracks which architectural registers have been modified by committed instructions since the last thread swap. The microcode sequencer uses this bitmask to selectively generate `rsave` instructions only for registers which have been modified. The correct value of unmodified registers is still in the IRB. For the short times that threads are often swapped in, this can significantly reduce the number of `rsave` instructions. Second, for those benchmarks which never use floating point registers, the floating point registers are not restored. Operating systems already use this technique to shorten software context switches. Both techniques shorten the time to swap threads and reduce contention for functional units with other active threads.

Duplicated Hardware—While registers are saved and restored on a context switch, some small bits of hardware can simply be replicated for each virtual context. These include the branch global history register, the return stack, and processor control registers, such as the page-table base register and floating-point control register. Each of these resources, which we expect are not likely to be on a critical circuit path, would need to be accessed through a multiplexer which would be controlled by a physical-to-virtual context mapping register. The special thread-switch instruction changes this register to correspond to the next thread to run.

Selecting the Next Thread—The next thread to swap in is known before a thread swap occurs. We use a Least-Recently-Run policy for selecting the next thread. When an active thread is swapped out of the pipeline, the least recently run thread is swapped in. It would be misleading to call this a Round-Robin policy. The order in which threads are swapped in can change over the course of a run because one thread may remain active for a long time, while several other threads are swapped in and out.

When a thread incurs a miss, but all inactive threads are also waiting for memory, we found that a good policy was to swap out the stalled thread, swap in the least recently run thread, but gate (stall) fetch for that least recently run thread until its data is returned from memory. This prevents the still-stalled thread from introducing instructions into the processor that will interfere with other active threads. Eickemeyer et al. [28] refer to this policy as *switch-when-ready* in their evaluation of a CGMT-only processor.

Inactive Register Buffer—Adding physical contexts to a processor increases the total number of registers in the register file, which is likely to affect the clock rate or pipeline length. The access requirements for active and inactive registers are quite different. As a result of these differences, the design constraints on the IRB are considerably relaxed, compared to the register file. (We will use the term *primary register file* to emphasize that we are not referring to the IRB.) For a 4-wide processor design, the IRB has at most 4 ports (read/write), compared to 12 ports (8 read and 4 write) for the primary register file. It does not require bypassing, because the same locations are never written and then read close together in time. Also, it can tolerate being placed far from the core pipeline, and thus has fewer layout constraints. In regard to the last item, we model a 10 cycle (pipelined) access time for the IRB, implying its distance from the core is similar to the L2 cache, certainly further than the L1.

In addition, firmware context switching is well-suited to a processor with a *unified register file* for both architectural registers and for uncommitted results, as in [128, 49]. In that type of architecture, including those with separate floating-point and integer register files, an architectural register is not mapped to a fixed location in the register file, so saving or restoring it involves first consulting the renaming table. The alternative architecture, with a separate reorder buffer and commit register file, may allow for greater hardware support of context switching, but it requires a higher read bandwidth on the reorder buffer for a given level of instruction throughput, and is poorly suited to SMT.

Our firmware approach to context switching does not add additional ports to the register file, since the thread switching operations use the ordinary instruction path. In summary, the inactive register buffer adds no complexity to the core of the processor.

VII.B.3 Time Required to Swap Threads

In our simulations, with the baseline BMT configuration, a majority of firmware context switches take 60 cycles or less. However, there is considerable room for variation. This section describes the range of times required for each step of the context switch.

25 cycles to detect main memory access—If a load instruction does not complete execution in 25 cycles, then it is considered to be a main-memory access. This includes a 3 cycle load instruction latency, a 14 cycle L2 latency, and several extra cycles to account for contention when accessing the L2 cache. This is for the baseline memory architecture. For the other memory designs investigated in Section VII.D.4, this threshold is adjusted. In principle, this time could be reduced by an early reply from the L2 tag array, or by consulting a load-hit predictor. However, as we show in Section VII.D.6, switching prematurely can decrease memory parallelism by missing the opportunity to issue independent load misses in parallel with the first miss encountered. The 60 cycle figure above does not include these 25 cycles.

3–30 cycles to trigger flush—There is a 3-cycle minimum delay to trigger a flush in our model. However, older uncommitted instructions from the same thread may further delay the flush. In our simulations, the flush occurs after 3 cycles 64% of the time, within 15 cycles 94% of the time, and very rarely after more than 30 cycles. A flush could be triggered before the canceled load becomes the oldest instruction in its thread, but we found that the cost of unnecessary flushes caused by wrong path instructions outweighed the advantage of flushing sooner.

15 cycles for microcode to reach execute—Instructions can be fetched from the microcode control store immediately after the flush has been triggered. In the pipeline we model, there are 15 stages between fetch and execute.

$\sim$10 cycles to issue `rsave` instructions—The microprogram will contain 1–62 `rsave` instructions, depending on the number of dirty registers. There is considerable variation between benchmarks. Overall, though, on 50% of thread swaps, 20 or fewer registers had been modified, and on 90% of thread swaps, 40 or fewer had been modified. The `rsave` instructions compete to use the integer units with instructions from other active threads, but in the best case, 40 `rsave`s take 10 cycles to execute, 4 at a time.

$\sim$16 cycles to issue `rrestore` instructions—The microprogram concludes with 62 `rrestore` instructions to restore the registers of the new thread. These take at least 16 cycles to execute. For those 4 of the 16 benchmarks which do not use floating-point registers, there are only 31 `rrestore`s.

$\leq$10 cycles restore-use latency—After the microprogram is fetched, but concurrently with the execution of the `rrestore`s, the processor fetches from the new thread. We model a 10 cycle latency for the `rrestore` instructions, and the execution of the `rrestore` instructions is fully pipelined. Depending on what

Fetch ≤ 3 instructions per thread from ≤ 2 threads each cycle
Branch prediction 64Kbit 2bcGskew **Deep Pipeline** 22 stages, 16 cycle misp. penalty
Out-of-order execution with 48/32/20 entry integer/fp/memory instruction queues, which may issue 4 integer/mem instructions (≤ 2 mem) and 2 fp instructions each cycle
Instruction Window supports 128 in-flight instructions†
Memory system
32k 4-way 3 cycle L1 Instruction and Data caches (2 acc/cyc)
64 byte linesize for L1 caches
64 entry DTLB / 48 entry ITLB, fully associative
256 entry second level Data and Instruction TLBs
128 byte linesize for higher-level caches
2MB 8-way 14 cycle L2 cache (1 acc/cyc)†
500 cycle memory access time†
†— Baseline parameter, different where noted.

Table VII.1: Processor parameters for Chapter VII.

registers are used first by the new thread, there will be a 0–10 cycle delay. This could be reduced by strategically reordering the `rrestore` instructions to match the order of their use by the new thread, based on the instructions previously flushed. Of course, the new thread may also incur an instruction cache miss.

VII.B.4 Common Architecture

The parameters common to all processor designs are shown in Table VII.1. We intend that these parameters represent a reasonable processor design one or two process generations from now, except that the cache sizes are somewhat smaller than might be projected. We chose relatively smaller cache sizes to match the memory footprint of the benchmarks we use.

The baseline SMT processors we evaluate implements the flush-on-cache-miss policy from [111], which makes more room in the instruction window for instructions from non-stalled threads. Thus, the miss detection and flushing capability required by BMT should not be viewed as an extra cost of our design.

We model a software TLB miss handler mechanism close to that used in the Alpha architecture [24] for all processor designs. For some workloads, page-table walks due to TLB misses represent a significant fraction of all main memory accesses, and a fraction which increases as more threads are run together.

Name	Code	Input	Fast Forward Instructions ($\times 10^6$)
ammp	0		2000
art	1	-startx 110	7500
crafty	2		700
eon	3	rushmeier	100
galgel	4		5000
gap	5		185330
gcc	6	166	2100
gzip	7	graphic	39300
mcf	8		12600
mesa	9		1300
mgrid	A		2100
parser	B		400
perl	C	makerand	10000
twolf	D		900
vortex	E	2	6000
vpr	F	route	36100

Table VII.2: The benchmarks used in this study.

Therefore, we allow thread swaps to occur on the loads in the TLB miss trap handler routine. A system with a hardware TLB handler should be able to accommodate thread-swapping as well.

VII.C Methodology

We evaluate each design alternative by simulation. For each design, we simulate workloads of different sizes. For each workload size, we present the average of several different workloads. Each of the workloads are comprised of a subset of the SPEC2000 benchmarks.

We perform all simulations using a detailed, execution-driven simulator, based on SMTSIM [113]. The simulator executes Alpha binaries which are compiled with the DEC C (`-O4`) or Fortran (`-O5`) compiler. We added a software TLB miss handler that closely models the Alpha architecture PALCode TLB trap handler.

The speedup results we present are meant to be an estimate of the overall improvement in throughput for a system which continuously runs the 16 benchmarks shown in Table VII.2, as compared to a single-threaded system. We

2A	01	Workload ↓ Name		4A	0123	8A	01234567
2B	12			4B	4567	8B	89ABCDEF
2C	23	3A	012	4C	89AB	8C	02468ACE
2D	34	3B	345	4D	CDEF	8D	13579BDF
2E	45	3C	678	4E	0246		
2F	56	3D	9AB	4F	8ACE	10A	0123456789
2G	67	3E	CDE	4G	1357	10B	456789ABCD
2H	78	3F	024	4H	9BDF	10C	89ABCDEF01
2I	89	3G	68A			10D	CDEF012345
2J	9A	3H	CEF	6A	012345		
2K	AB	3I	135	6B	6789AB	12A	456789ABCDEF
2L	BC	3J	79B	6C	ABCDEF	12B	012389ABCDEF
2M	CD	3K	DF6	6D	0369EF	12C	01234567CDEF
2N	DE			6E	147C28	12D	0123456789AB
2O	EF	Bench.↑ Codes					
2P	F0			(see Tbl VII.2)		16A	01234567 89ABCDEF

Table VII.3: The workloads used in this study. (Refer to benchmark codes in Table VII.2.

simulate a portion of each benchmark. With the assistance of SimPoint [98], we select a starting point for simulation within each benchmark. Using the multiple simulation point algorithm, we select a phase in each benchmark that represents the largest amount of execution.

We simulate several different workloads for each workload size, which represent a sampling of the space of possible workloads. The exact combinations used are shown in VII.3, where each workload is described as a string of characters. Each character represents a benchmark, as shown in the column labeled *Codes* in Table VII.2. For example, workload `2B` consists of 2 threads, `art` and `crafty`. The workloads are selected so that each benchmark is included in more than one workload at each workload size, and to reduce commonality between workloads without unduly increasing the number of simulations. Beyond that, the combinations are selected without any design.

In all simulations, after advancing each thread to the simulation starting point indicated in Table VII.2 using a checkpoint, we performed a detailed simulation until $10^8 \times n$ instructions had been executed (where n is the number of threads in the workload). When simulating multiple threads, each benchmark in a single workload will run for a different number of instructions under different processor parameters. If there is a large variation in performance of the running threads, this will complicate interpretation of the results. Thus, we present all performance results as *weighted speedup* [102, 111]. The weighted speedup of a multithreaded workload is defined as the sum of the speedups of each individual thread within the workload over a baseline run (in this case single-thread execution). The speedup of a thread within a workload is defined as its performance, in instructions per cycle (IPC), when part of a multithreaded run, divided by its IPC when run by itself over the same range of instructions. Thus weighted speedup represents average relative progress on the workload. By contrast, other metrics, like total instructions per cycle, can artificially create the appearance of increases in performance when more instructions are executed from a higher-IPC thread.

Each speedup we present for 2, 3, 4, 6, 8, 10 or 12 threads at a time represents the average of 16, 11, 8, 6, 5, or 4 different simulations, respectively, as shown in Table VII.3.

We use CACTI 3.2 [99] for modeling register access times. Since CACTI 3.2 is designed to evaluate the access time of caches, we discarded the tag path in the measures presented here. Access times assume a 70nm process.

VII.D Analysis and Results

The number of physical contexts supported by a processor, C_{phys}, affects the size of the physical register file and the renaming table, both of which are likely to affect the maximum clock speed and pipeline length. This can degrade performance, especially when one or few threads are run at a time. In this

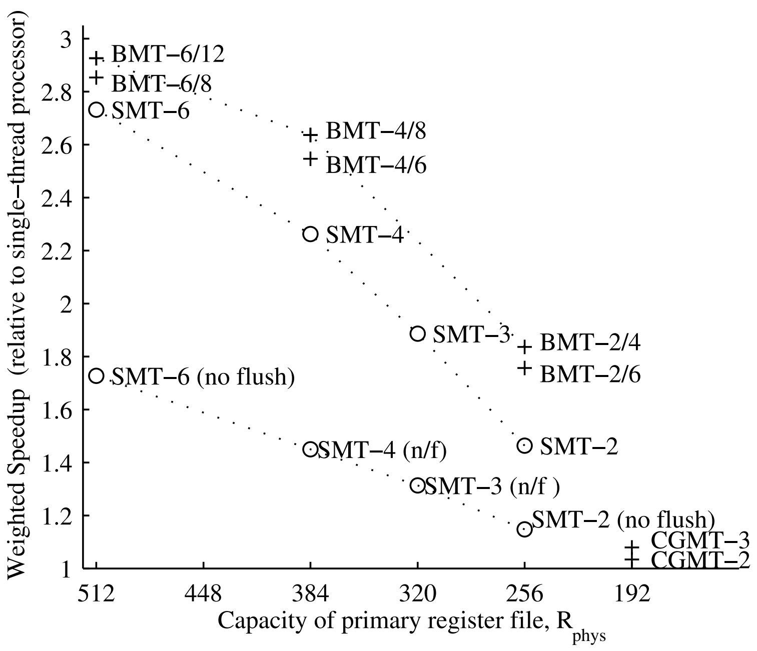


Figure VII.1: Performance (weighted speedup) of SMT, BMT and CGMT vs. physical register file size.

paper, we use the number of physical registers, R_{phys}, as a proxy for these other effects. The following subsection examines the relationship between throughput and R_{phys} for different multithreading schemes. Also in this section, we examine how BMT performs over a range of workload sizes and memory parameters. We also examine the importance of firmware support for thread switching, and of store retirement policies. We consider the effect of changing the delay to trigger a thread swap after a miss. Finally, we quantify the effect of a larger register file on overall performance.

VII.D.1 Increasing Throughput Simply

Figure VII.1 illustrates the tradeoff between throughput and physical register file size. The figure shows that, for a given register file size, a BMT processor gets greater throughput than an SMT processor.

The x-axis shows the number of registers in the physical register file, R_{phys}. We define

$$R_{phys} = C_{phys} \times R_{arch} + R_{ren}$$

where $R_{arch} = 62$ because the Alpha ISA defines 62 non-zero registers, and, again, C_{phys} is the number of physical contexts. All designs assume a single unified physical register file. The IRB is not included in R_{phys}, because it should not be part of a critical circuit timing path. For all the results, except where noted in Section VII.D.7, $R_{ren} = 128$, which permits 128 in-flight instructions across all threads. The y-axis shows speedup relative to an otherwise equivalent single-thread processor.

There are 4 groups of points to consider. The points labeled *SMT-n*, in the middle curve, show the speedup of SMT processors running a workload of size n. For an SMT processor, $n \leq C_{phys}$, and $C_{phys} = C_{virt}$, so we use $C_{phys} = n$ to compute R_{phys}. The points labeled *SMT-n (no flush)*, in the lower curve, show the performance of a series of SMT processor designs without a mechanism to flush a thread with a long-latency load [111]. We present this to emphasize the importance of having such a mechanism in any multithreading processor with a shared instruction window. The points labeled *BMT-m/n* represent BMT designs with $C_{phys} = m$ and $C_{virt} \geq n$, running workloads of n threads.

The *BMT-2/4* processor gets 26% more throughput than an *SMT-2* processor, while running at the same clock speed with the same pipeline depth. A 4-context SMT processor gets 17% more throughput when enhanced with BMT, assuming 8 jobs are ready to run.

We model the same pipeline depth and cycle time for all SMT and BMT configurations. As additional hardware contexts are added, keeping the pipeline depth or cycle time constant is unlikely, but the focus of our comparisons are between SMT and BMT configurations with the same number of physical contexts. Because the speedup is not adjusted for these effects, care should be taken when comparing points with different values of R_{phys}. For example, while the 4-context SMT processor shows 54% higher throughput than a 2-context SMT processor when 4 threads are available, differences in the pipeline and/or clock rate between those two designs mean that the relative throughput of the 4-context SMT processor will be lower than that number.

Even ignoring complexity differences, however, the additional benefit of our approach is significant. Regardless of whether a 2, 4 or 6 context SMT processor design is the best choice for particular technology and performance goals, BMT can be added to boost throughput without affecting pipeline complexity. Additionally, these results assume all physical contexts are filled. When there are fewer threads than contexts, the advantage of the BMT designs over SMT are even greater.

This figure also shows the performance of CGMT alone. It provides only marginal gains over a single-threaded processor. Because of the high cost of moving state in and out of the processor core, CGMT alone is of less value. But when CGMT is added to SMT, the additional physical contexts can do useful work while a context switch is underway, hiding the cost of the switch.

VII.D.2 Scalability of Balanced Multithreading

Adding more threads to a processor can increase performance by increasing memory parallelism. However, with too many threads, the benefits can be outweighed by the cost of contention between threads. In this section, we investigate how well different BMT designs perform, compared to SMT designs, as the virtual-to-physical context ratio, C_{virt}/C_{phys}, increases.

The firmware mechanism to swap threads in and out of the processor core has two costs. First, the time required to complete the context switch delays the start of execution of the incoming thread. Second, the firmware save/restore instructions contend with other active threads for execution resources. To understand the cost of the firmware context switching mechanism, we compare the performance of the firmware mechanism with a hypothetical *instant* save/restore mechanism.

Figure VII.2 shows the weighted speedup of several different SMT and BMT designs. The x-axis shows the number of threads in a workload, n, which is assumed to be equal to C_{virt} for this study. The y-axis shows weighted speedup of each design compared to a single-thread processor. On the curve where the points are labeled *SMT-n*, the points represent SMT processors capable of running workloads of n threads together. There are three sets of curves for BMT designs with 2, 4, or 6 physical contexts. Within each set, there is a curve labeled *firmware*, for a processor using the firmware thread swapping mechanism, and a curve labeled *instant* which represents a processor with an idealized, nearly instantaneous thread-swapping mechanism. The instant mechanism requires only 1 cycle to save and restore the architectural registers of the outgoing and incoming threads, once the miss-to-memory is detected and a thread is flushed.

Figure VII.2 illustrates two effects. First, for each value of C_{phys}, there is an optimal value of C_{virt}/C_{phys}. Second, as C_{phys} increases, the relative cost of the firmware thread swapping mechanism increases too. The figure shows that the gain from BMT peaks when $C_{virt}/C_{phys} = 2$. When the ratio is larger than 2, the costs of running multiple threads begin to outweigh the benefits. For a BMT processor, that cost has two components: the cost of thread swapping and the cost of interference between threads. The curves labeled *instant*, while being perhaps impractical, show the relative contribution of these two effects. When n is small, the cost of swapping is low. The cost of thread swapping comes from contention for instruction queue space and load/store ports from the thread-

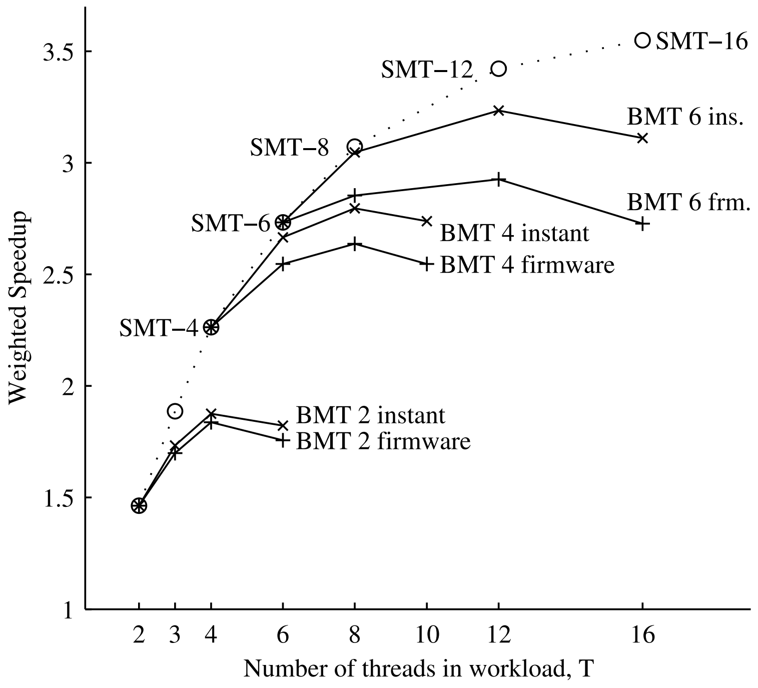


Figure VII.2: Weighted speedup of several different SMT and BMT designs. The X axis shows the number of scheduled virtual contexts for the processor configuration; that is, the total number of threads run together at once.

swapping instructions. Thus, at the *BMT-2* design point, there is little reason to try to further optimize the thread swapping mechanism, but for *BMT-6*, there is an incentive to improve it.

For larger values of C_{virt}/C_{phys} and larger n, the benefit from increased memory parallelism is outweighed by a loss of locality in the higher level caches. The loss of locality is caused by having many threads in the workload. The optimal C_{virt}/C_{phys} ratios suggested by this graph are for an average over many workloads, but will vary with the particular threads running. This represents an opportunity to further improve performance by adaptively sizing the number of threads in a workload based on the behavior of the constituent threads.

VII.D.3 Hardware Support for Thread Swapping

The previous section compared the performance of our baseline thread swapping mechanism with a hypothetical one-cycle latency thread swapping mechanism. Our baseline mechanism already includes some optimizations to reduce swapping latency. This section evaluates two of those optimizations: the Dirty Register Mask (DRM) and the Inactive Register Buffer (IRB).

The DRM, discussed in Section VII.B.2, allows the thread swap to only save registers values that have been touched. The IRB may be considered an optimization compared to a purely software thread swap, where a context's state is stored using conventional loads and stores. Figure VII.3 shows the performance of BMT processors with 2 or 4 physical contexts, with varying levels of hardware support for thread swapping, and of SMT processors with 2–6 contexts.

The two BMT features are not important for the *BMT-2* processor, but are important for the *BMT-4* processor. Of the two, the DRM is more important. The benefit of the dirty-register mask increases as more threads are run because of the greater contention for functional units. A lesser effect may be that programs

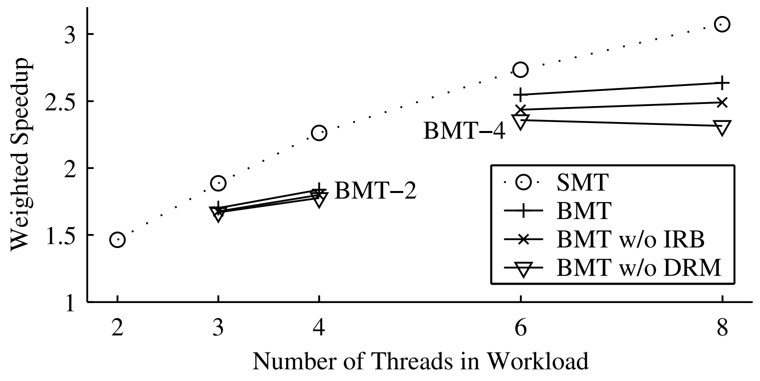


Figure VII.3: Performance of BMT with different levels of hardware support.

are swapped in for less time when more threads are present, and thus have time to dirty fewer registers.

Without an IRB, inactive registers could be stored directly into memory (where they would typically be caught by the cache). Thus, for the no-IRB configuration, the save-restore instructions use the load/store units, which halves the rate at which they may issue. In the no-IRB configuration, if a miss occurs in the thread-swap microcode, the thread waits instead of performing a second swap. Because such misses are uncommon in the *BMT-2* configurations, there is little performance impact. With a larger workload size, the IRB is important for good performance.

VII.D.4 Sensitivity to Memory Hierarchy

The speedup provided by balanced multithreading is sensitive to three parameters of the memory hierarchy: The size of the caches, the latency to access the lowest level of cache, and the latency to main memory. Figure VII.4 shows the performance of SMT and BMT with different memory configurations. Each group of bars shows the performance of different processor designs with the same memory hierarchy. All configurations have the L1 caches described in Table VII.1, but the lower levels of the hierarchy are varied. The configurations were chosen to study the sensitivity to individual memory-system parameters. The y-axis represents weighted speedup. For each group of bars, the speedup is computed relative to a single-threaded processor with the same memory hierarchy. As a result, the speedup for a design with a larger cache hierarchy may be less than that for a design with a smaller cache.

The best C_{virt}/C_{phys} ratio for a BMT system depends on the memory system, so we show two BMT configurations next to each corresponding SMT processor design. Above each group of bars is shown the speedup of the better of the two BMT bars over the adjacent SMT bar. All three of those bars the same C_{phys}. For example, the first group of bars, labeled *Base*, represents the memory configuration used for all previous results in this paper: a 500 cycle memory latency and a 14-cycle 2MB L2 Cache. As noted in the plot, a *BMT-2/4* design gets 26% speedup over an *SMT-2* processor, and a *BMT-4/8* gets 16% speedup over an *SMT-4* design.

The speedup from applying our technique will be sensitive to three parameters of the memory hierarchy: The size of the caches, the latency to access the lowest level of cache, and the latency to main memory. Figure VII.4 shows the performance of SMT and BMT with different memory configurations. Each group of bars shows the performance of different processor designs with the same memory hierarchy design. All configurations have the L1 caches described in Table VII.1, but the lower levels of the memory hierarchy are varied as noted in the figure. The configurations were chosen to study the sensitivity to individual memory-system parameters. The y-axis represents weighted speedup. For each group of bars, the speedup is computed relative to a single-threaded processor with the same memory hierarchy. As a result, the speedup for a design with a larger cache hierarchy may be less than that for a design with a smaller cache.

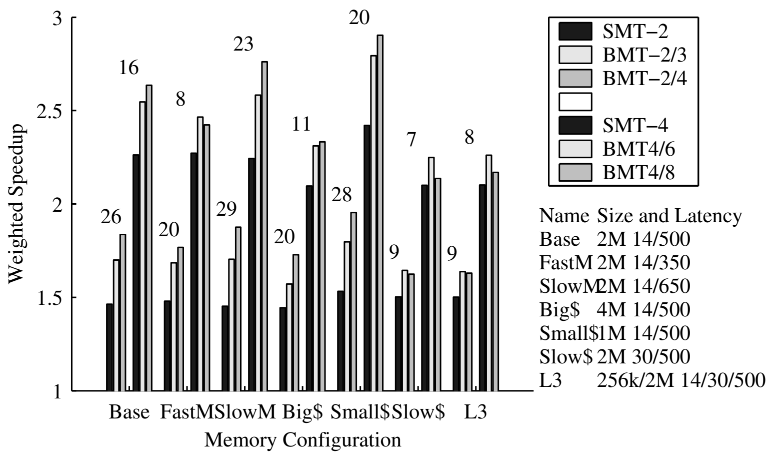


Figure VII.4: Performance (weighted speedup) of SMT and BMT for several different memory hierarchies.

The best C_{virt}/C_{phys} ratio for a BMT system depends on the memory system, so we show two BMT configurations next to each corresponding SMT processor design. Above each group of bars is shown the additional speedup of the better of the two BMT bars over the adjacent SMT bar, all of which have the same C_{phys}. For example, the first group of bars, labeled *Base*, represents the memory configuration used for all previous results in this chapter: a 500 cycle memory latency and a 14-cycle 2MB L2 Cache. As noted in the plot, a *BMT-2/4* design gets 26% speedup over an *SMT-2* processor, and a *BMT-4/8* gets 17% speedup over an *SMT-4* design.

Running more threads at the same time has a cost and a benefit. Part of the cost is from increased contention in the caches, predictors and other structures. The benefit is an increase in the number of parallel memory accesses. Changes to the memory parameters shift these costs and benefits.

A larger cache, as in *Big$*, reduces the number of opportunities to use coarse-grained thread switching. Also, a slower cache increases the cost of misses caused by cache contention, and increases the latency before the processor can detect a main memory access. This is illustrated by the lower additional speedup from BMT for the *Slow$* group.

The *L3* configuration has a third level of cache, which has both the detrimental effects just mentioned. In this configuration, context switching only occurs on an L3 miss, because the firmware context switch mechanism is too slow to hide an L3 hit. With a faster memory (*FastM*), the fraction of time spent on context switches relative to total execution time increases.

In the case of the larger cache for the *Big$* configuration, there are simply not enough misses to main memory to offset the increase in contention. For example, the *BMT-4/6* processor with the Base memory configuration had, over all workloads, 2.5 main memory accesses per 1000 committed instructions. The same processor with the *Big$* memory configuration only had 0.8 main memory accesses per 1000 instructions. The larger cache significantly reduces the opportunity to benefit from thread swapping. At the same time, the number of Data Cache misses which do not go to memory increases. For *BMT-4/6*, with the *Base* memory, there are 16 L1 misses that do not go to memory per 1000 instructions. For the *Big $* configuration, there are 22 L1 misses per 1000 instructions that are filled without going to memory.

It should be noted that the lessened need for BMT with large caches is primarily a function of the workload, rather than the architecture. Even today, many commercial applications will exercise caches of this size much more heavily. Thus, while cache sizes will increase, which reduces the number of main memory accesses, which in turn reduces the effectiveness of our technique, we expect this effect to be largely mitigated by increases in application working set sizes. Thus, when evaluating our technique, we feel it is fair to focus on the results for the baseline memory configuration.

The bar-groups labeled *FastM* and *SlowM* show results for processors with 350 and 650 cycle main memory latencies, respectively. We use 500 cycles

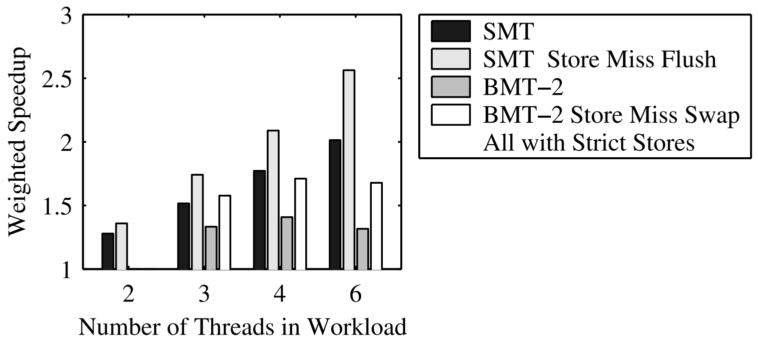


Figure VII.5: Performance of SMT and BMT processors with strict store retirement policy.

as the baseline main memory latency, and we expect that real systems will reach that level soon. As memory latency increases, the advantage of adding more virtual contexts increases: With *SlowM*, the BMT2/4 and BMT4/8 configurations significantly outperform BMT2/3 and BMT4/6, respectively.

VII.D.5 Store Retirement Policies

All the architectures presented in this chapter allow store instructions which miss in cache to *partially complete*; younger non-store instructions may commit, and free up space in the instruction window, even when a store's result has not yet been written to the L1 data cache. We believe that this fairly reflects some modern processor designs. Nevertheless, we also considered an with a *strict* store retirement policy; younger instructions wait for a store to write to the L1 cache. A strict store retirement policy might be necessary in some systems to insure timely handling of interrupts. With a strict retirement policy, a long-latency store may cause a thread to fill up the instruction window, stalling progress for all threads. ITo counteract this, we found that swapping on long latency stores as well as loads produced good results. Figure VII.5 evaluates SMT and BMT architectures with a strict store retirement policy. This shows that, under a strict

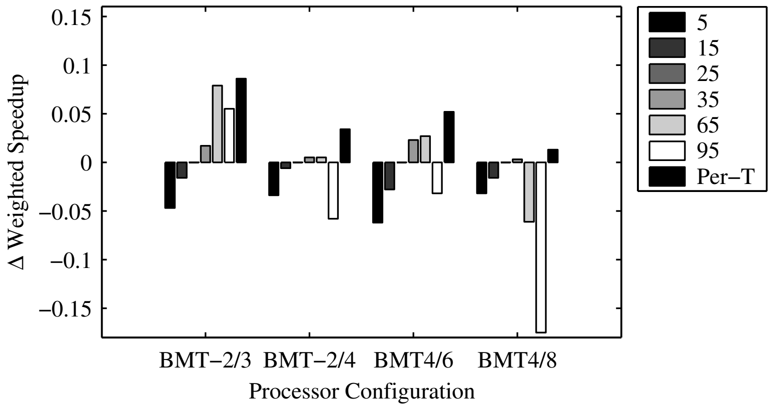


Figure VII.6: Performance of BMT processors with different delays to initiate swapping on a miss.

store retirement policy, SMT and BMT architectures benefit significantly from flushing/swapping on stores, and that BMT works well with strict stores as well.

VII.D.6 Delayed Detection of Load Misses

In the baseline BMT configuration, a thread swap is triggered when a load instruction takes longer than 25 cycles to complete. The minimum latency for an L2 hit in the baseline architecture is 17 cycles, but some loads take longer due to contention at the L2 cache. Waiting an additional 8 cycles avoids premature swapping. The simple wait-25-cycles approach only requires a small counter for each active load instruction. A alternative mechanism might include a signal from the L2 cache after the tags has been checked. Detecting a load miss and switching sooner may improve performance, since the next thread begins executing sooner. However, flushing a thread too soon can prevent the execution of a second load instruction, which would otherwise initiate a second, parallel, main-memory access.

We evaluated the performance of 4 BMT designs with different values for the load-execution to miss-detection latency, l. Those results are shown in Figure VII.6. The value of l is indicated in the legend. The y-axis shows the

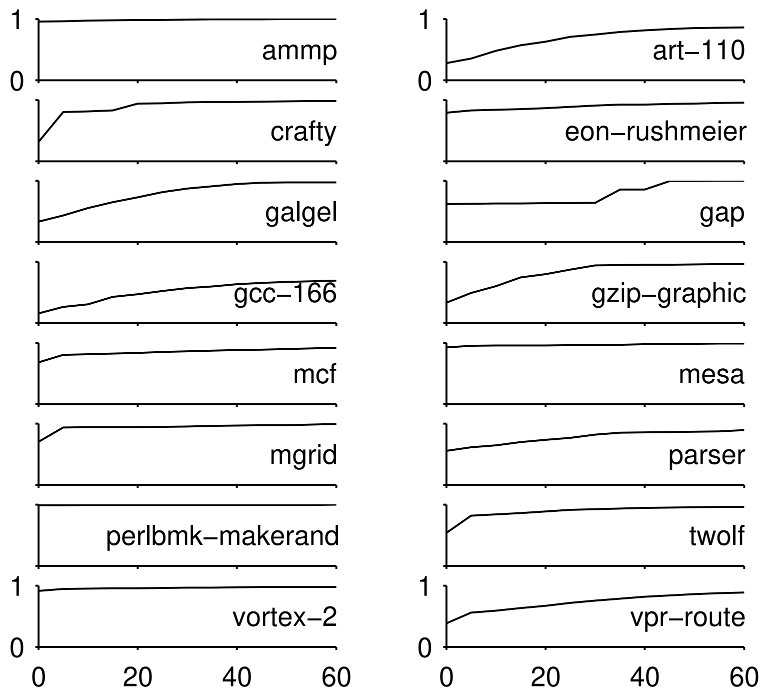


Figure VII.7: Probability, y, at time x after executing a load instruction which misses in L2, that no further main-memory accesses will be initiated.

change in weighted speedup for a given design when l is changed from its baseline value of 25. Note that detecting an L2 miss after only 5 cycles would require either checking the L2 tags very quickly, or a load hit predictor. Fortunately, detecting a miss sooner actually decreases throughput. For example, if L2 misses could be detected 5 cycles after a load first executed, the weighted speedup of *BMT-2/3* would drop by 0.05 (from 1.70, as indicated in Figure VII.1 or Table VII.4, to 1.65).

In all cases, increasing l to 35 increases the throughput of BMT. However, with larger workloads, higher values of l may reduce throughput. With a larger workload, it is more likely that there is a ready-to-run thread waiting to be swapped in. The best single value of l depends on the number of virtual and physical contexts, and the particular set of benchmarks. However, an even better policy would be one which sets a different value of l for each thread.

It is profitable to delay swapping out a thread if it is likely that additional main memory accesses can be initiated by waiting. As illustrated in Figure VII.7, benchmarks differ considerably in the number of main-memory accesses that may occur in parallel. There is one subgraph for each benchmark we use. The y-axis shows the probability that no additional main-memory accesses will be initiated following a load which misses in the L2 cache. The x-axis shows time in cycles after the first miss. Note that only subsequent accesses to different cache lines are counted. For `perl` amd `ammp`, when a load misses in the L2, it is highly unlikely that subsequent loads will initiate additional memory activity, so those threads should be swapped out as soon as possible. For `gcc`, even 60 cycles following a L2 miss, it is quite likely that additional misses will occur before the first miss completes, so `gcc` should be swapped out after a longer delay. We evaluated a static, per-thread swap-delay policy. This is shown as the bar labeled *Per–T* in Figure VII.6. For this policy, all threads are swapped out on

Type	C_p	n	WSU	R_{ren}	R_{phys}	t_{acc}	n_{stg}
Uni	1	1	1.00	128	190	0.46	5
SMT	2	2	1.46	128	252	0.58	6
	3	3	1.89	”	314	0.60	7
	4	4	2.26	”	376	0.62	7
	6	6	2.73	”	500	0.65	7
	8	8	3.07	”	628	0.72	8
BMT-2	2	3	1.70	128	252	0.58	6
	2	4	1.84	”	”	”	
	2	6	1.76	”	”	”	
BMT-4	4	6	2.57	128	376	0.62	7
	4	8	2.64	”	”	”	
BMT-6	6	12	2.93	128	500	0.65	7

C_p	the number of physical contexts
n	the number of threads in workload
WSU	the weighted speedup
R_{ren}	the number of additional registers for renaming
R_{phys}	the number of physical registers
t_{acc}	the register file access time (ns)
n_{stg}	the estimated number of stages at 10 Ghz

Table VII.4: Performance and expected register file access times for various multithreading architectures.

a load which takes more than 20 cycles, except `art`, `galgel`, `gap`, `gcc`, and `vpr`, for which $l = 80$. In all 4 cases, the *Per–T* policy performs better than any single value of l. With this policy, BMT2/4 gets an additional 3% speedup over single-thread execution.

We present the Per-T policy to show that there is benefit from a dynamic policy which detects which threads have high memory level parallelism. To implement such a policy, l could be held in a counter which is periodically set to a high value, and which is decremented each time no concurrent misses occur.

VII.D.7 Quantifying the Cost of Additional Registers

The weighted speedup results presented in this paper do not reflect any cycle time or pipeline length penalties that may arise from adding physical contexts to a processor. In this section, we attempt to quantify the cost of adding

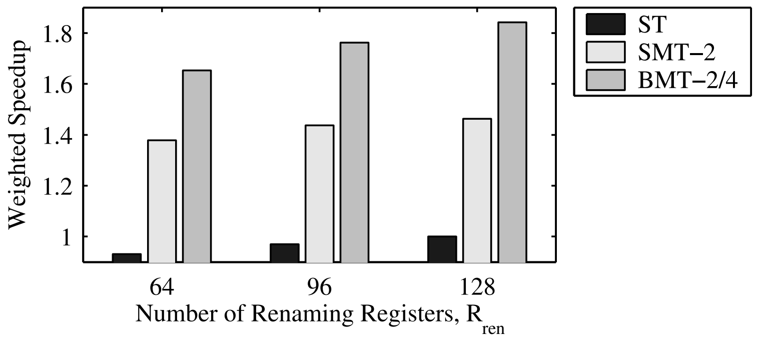


Figure VII.8: Performance (weighted speedup) of SMT and BMT for several different numbers of additional renaming registers over the baseline 128 logical registers needed. All speedups are relative to single-thread processor with 128 registers provided for renaming (190 total).

additional physical contexts to an SMT processor, as opposed to adding virtual contexts.

Table VII.4 lists the different architectures studied in previous sections. The speedups shown are for the base memory configuration (see Table VII.1). The last two columns show estimates of the register file access times for different architectures and an estimate of the number of clock cycles that it would require if pipelined at 10 GHz. By this estimate, 3 additional pipeline stages would be needed for an 8 context SMT processor, compared to an otherwise similar 1-context processor. Our access time estimates do not quantify several additional costs of additional contexts. A slower register file read time can add stages between issue and execute, which complicates scheduling. A slower register file write time requires additional hardware to hold bypassed results longer. And a larger register file in turn increases the size of the renaming table. Also, the additional pipeline stages required to tolerate a larger register file fall in a particularly inopportune place in the pipeline. Lengthening the pipeline at this point increases load hit misspeculation penalties [10].

In previous sections, we simulate processors with a large instruction window. The instruction window requires 128 registers beyond those required to hold programmer-visible state (which is 62 per physical context). An alternate way to reduce the size of the register file is to provide fewer of these additional registers. Doing this does not negate the benefit of BMT. If reducing the size of the instruction window increases the performance of the processor, or makes room for additional physical contexts, then BMT can still be used. Figure VII.8 shows that a *BMT-2/4* processor configuration beats an SMT-2 processor configuration, with fewer additional registers for renaming (a smaller instruction window).

VIII

Summary and Future Work

VIII.A In This Dissertation

Modern processors remove many artificial constraints on instruction ordering, permitting multiple instructions to be executed in parallel. As a result, only a fraction of all the instructions in a program trace determine the execution time of the program. Any effort to improve program performance is wasted when not applied to these *critical* instructions. Likewise, the remaining *non-critical* instructions may be delayed, to a point, without affecting performance. Depending on the program and microarchitecture, typically between a few percent and half of all dynamic instructions are critical. We propose and evaluate several hardware techniques to classify whether an instruction is critical or non-critical, and discuss related efforts at the same. We show that the criticality of dynamic instructions is correlated to the corresponding static instruction. We exploit this correlation to predict an instruction's criticality, in hardware, before it executes. We call this *critical-path prediction.* These predictions can be used anywhere that the processor must arbitrate between instructions for a limited resource. We demonstrate the utility of these predictions in several such applications, which we call *critical-path aware optimizations*: a processor with a limited-rate value-predictor, a clustered microarchitecture with inter-cluster communication delays, and a reduced-power microarchitecture with heterogeneous functional units and queues. We perform an offline analysis of the critical paths of programs to validate our findings and to quantify the degree of criticality of different instructions. Our findings lead us to propose a new multithreading architecture. Under our proposal, threads execute in parallel in a manner sensitive to the hardware implications of supporting multiple contexts, and also sensitive to the critical path issues.

VIII.A.1 Critical Path Prediction

This thesis introduces the concept of critical path prediction, which seeks to identify those instructions that constrain the performance of the processor. We propose several critical path predictor designs.

Our first proposal is heuristic critical path prediction. This scheme relies on the behavior of individual instructions as they pass through the pipeline to indicate which instructions are critical. One of the most effective heuristics relies on the instruction scheduling mechanism already present in the processor to indirectly indicate relationships between instructions. This focus on individual instructions means that an implementation of a heuristic critical path predictor may require the least hardware.

The iterative predictor starts by identifying instructions which are definitely on the critical path, and then builds on the resulting predictions to gain a broader view of the critical path. The hybrid predictor combines the iterative predictor and a heuristic predictor for more consistent good performance over a range of benchmarks.

We also compare our predictors to a proposal by another research group: the token-passing predictor. We find the additional complexity of the token-passing predictor does not bring a clear advantage in comparison to our predictors.

All the critical path predictors rely on the assumption that static instructions which were recently critical will be critical again on subsequent iterations. We use a critical path prediction buffer to accumulate observations of critical instructions, and, in turn, to predict the criticality of instructions.

VIII.A.2 Critical Path Aware Optimizations

Critical path prediction information can be useful just about anywhere the processor must arbitrate between instructions, or where hardware structures are prone to contention or pollution. We demonstrate the utility of critical path prediction by using it to control three different aspect of a processor. We demonstrate that the effectiveness of a value predictor can be more than doubled through the use of critical path prediction, relative to a value predictor that must select randomly among multiple instructions that are deemed to be predictable. It is 68% more effective than a value predictor that uses decoded instruction information to make the selection based on expected latency.

We also demonstrate how critical path predictions can be used to steer instructions in a clustered architecture, where there is a communication latency between replicated pipeline stages. The effect of this additional inter-cluster communication latency is mitigated when critical instructions are sent to one cluster, and non-critical instructions are sent to a different cluster. We find that critical-path predictions improve on previous proposals for steering instructions, which did not take criticality into account.

It has also been shown that critical-path predictions can be used to reduce power consumption without excessive performance loss. Critical-path prediction can be used to direct instructions to execute on one of several functionally equivalent execution paths. These paths differ, however, in the power/performance tradeoffs inherent in their design. The viability of such a design demonstrates that critical-path prediction not only improves existing microarchitectures, but also makes possible new ones.

As concurrecy in processors continues to increase, application performance becomes more tied to the execution of the critical dependence path. These and other critical path aware optimizations will have an increasingly large advantage.

VIII.A.3 Quantifying the Critical Path

Having demonstrated the feasability of dynamic critical path prediction with several applications, we turn to an offline analysis of program traces to help use better understand the critical paths of programs. We analyze the criticality (tautness) or non-criticality (slack) of each instruction in a program trace. The former, tautness, is a new metric we introduce to measure the degree of criticality of a dynamic instruction. We evaluate the accuracy of several critical path predictors, both in terms of the accuracy of the training stream and the prediction mechanism; and we demonstrate the potential for new prediction techniques. We find that there are more slackful instructions than taut instructions, and examine the distribution of slack and tautness in programs. We found that a majority of static instructions are never critical, but among those static instructions that are ever critical, criticality varied frequently – very few static instructions are always critical; about 1.5% on average over 13 benchmarks. Thus, predicting exactly the dynamic instances of these static instructions that are critical is difficult, but important for a highly accurate predictor. We show that new prediction techniques that recognize patterns in these critical instructions have the potential to significantly increase the accuracy of critical path predictors.

Our work suggests several important future directions to improve the effectiveness of critical path prediction. It shows that critical path predictors need to be able to identify patterns of criticality to achieve higher coverage and accuracy. It also demonstrates the need for predictors that quantify criticality (or slack) rather than just produce a binary prediction. We provide one example appliction: selecting loads to speculatively precompute.

VIII.A.4 Multithreading

We also propose a new for multithreading which combines Simultaneous Multithreading (SMT) and specialized form of Coarse-Grained Multithreading (CMGT). We call the combination *Balanced Multithreading*. SMT allows the processor to tolerate even the smallest latencies. CGMT is sufficient to tolerate long memory latencies. We present a form of CGMT which requires no changes to timing-critical processor resources such as the register file and the renaming table. The combination of the two results in a processor that provides high single thread performance via a high clock rate, shorter pipeline and high instruction-level parallelism; and high memory parallelism and thread-level parallelism when more threads are available.

We find that in the face of long memory latencies, balanced multithreading can provide instruction throughput similar to a wide SMT processor, but without many of the hardware costs. In particular, we show that by adding support for balanced multithreading, the throughput of an SMT processor can be improved by 26%, with no significant changes to the core of the processor, the cycle time, or the pipeline.

While the work on Balanced Multithreading stands on its own as a valuable architectural technique, it is closely related to our work on critical-path prediction. Our finding that many load instructions which miss have a high tautness, in Chapter VI suggests that it is safe to delay all subsequent instructions for a long time while the load miss completes. And, we show that a critical-path related policy to control context switches further improves the performance of BMT.

VIII.B Future Work.

There is considerable opportunity to build upon and extend the ideas in this thesis.

Althogh we evaluate several critical path optimizations, and other researchers have evaluated several more, there is certainly room for even more. Using critical path predictions in several places in a processor has the additional advantage of amortizing the cost of the predictor.

Critical path predictions can be used to control whether the tag and data lookups in a set-associative data cache are performed in series or in parellel. Critical instructions would access in parallel for high-speed, and non-critical instructions would access in series for low-power. (When the two are accessed in parallel, all ways of the data array have to be accessed, but in series, only one way needs to be accessed.)

The critical-path aware optimizations evaluated or discussed in this thesis work by reducing the execution latency of instructions on the critical path. But several optimizations are possible which do not work by shortening execution latencies, but by optimizing the rate at which instructions enter the pipeline in accordance with the critical path. Different programs may enjoy more or less benefit from keeping a larger instruction window full of instructions. If there are many critical instructions waiting to execute, this suggests that the processor does not need to fetch further ahead until some more critical instructions execute. This could be an opportunity to turn off an unused portion of the instruction queue, or to throttle the fetch and decode stages in a power-saving way.

There are several directions for inquiry which combine multithreading and critical-path prediction. First, critical path prediction can be used to improve resource sharing in an SMT processor. In an SMT processor, the instruction queues, and physical register file may be shared between threads, either by a partitioning, or by a fetch priority mechanism. A critical path aware window partitioning or fetch priority policy could result in a better division of resources than one that counts critical and non-critical instructions the same. Second, some previously examined critical path aware optimizations may be more attractive in the context of a multithreading processor. For example, the criticality-controlled victim cache considered by Srinivasan et al. might be more useful in a multithreaded processor, where the level of cache contention is higher. Instruction scheduling may be more attractive in a multithreaded processor as well.

Of course, the critical-path predictors will need to be adapted somewhat to accomodate multiple threads. For several non-interacting processes, we expect the changes will be quite limited. For multi-threaded shared memory programs, additional work needs to be done. How can we track the critical path as it jumps between several threads? If a load of a shared memory location is on the critical path for one thread, then the instructions leading up to the store of that item in another thread should be considered critical as well. Some preliminary research has already begun in this area, [68]. One result is that it is possible to view the sections of code between synchronization instructions as a single node in an inter-thread dependence graph. Under this scheme, the entire thread would be designated as critical over the span of, say, hundreds or thousands of instructions. This suggests a range of thread-segement-specific as opposed to instruction-specific optimizations. One such scheme is analagous to the multi-speed functional units optimization described in Section V.C. Each thread could be directed to execute on one of the cores of a *heterogeneous multi-core processor*, [65], as appropriate to its criticality. The critical thread would execute faster, which the non-critical threads would execute on lower power processor cores. Even in a chip-multiprocessor which is not heterogeneous, the power consumption of indivual cores could be adjusted.

For application-specific processors [43] and reconfigurable processors [46], hardware is designed, or configured, to take advantage of the properties of a single program or class of programs. To the extent that those programs vary at runtime, dynamic critical-path prediction could be used to further optimize the behavior on such architectures. In the case of reconfigurable processors, runtime critical path profiling could be used to guide reconfiguration of hardware, when precise program behavior is not known at compile/design-time.

Because the level of parallelism that processors can expose and exploit continues to increase, both at the instruction level, and at the thread level, the benefits from critical path aware optimization will grow as well. And because designers have an increasing number of transistors at their disposal, even more sophisticated program analyses may be possible in hardware.

Bibliography

[1] *Computer Industry Almanac.* Computer Industry Almanac, Inc., Eighth edition.

[2] Alpha 21264/EV6 Microprocessor:Hardware Reference Manual. Compaq Corporation, 1998.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[4] Cedell Alexander, Donna Reese, and James Harden. Near–critical path analysis of program activity graphs. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 308–317. IEEE Computer Society, February 1994.

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[6] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11:8–24, January 1967.

[7] R. Iris Bahar, Gianluca Albera, and Srilatha Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 64–69, New York, August 10–12 1998. ACM Press.

[8] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *In Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[9] A. Battersby. *Network Analysis for Planning and Scheduling.* Wiley, 1970.

[10] Eric Borch, Eric Tune, Bobbie Manne, and Joel Emer. Loose loops sink chips. In *Eigth International Symposium on High Performance Computer Architecture*, February 2002.

[11] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, July-August 1999.

[12] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kallaa, and S. R. Kunkel. A multithreaded powerPC processor for commercial servers. *IBM J. Res. Dev.*, 44(6):885–898, 2000.

[13] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA1999*, January 1999.

[14] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *HPCA2001*, January 2001.

[15] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[16] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction steam parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture*, pages 276–286, May 1991.

[17] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *MICRO33*, December 2000.

[18] Alper Buyuktosunoglu, David H. Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. Power-efficient issue queue design. pages 35–58, 2002.

[19] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–75, May 1999.

[20] Jason Casmira and Dirk Grunwald. Dynamic instruction scheduling slack. In *2000 KoolChips workshop*, December 2000.

[21] Francisco J. Cazorla, Enrique Fernandez, Alex Ramírez, and Mateo Valero. Improving memory latency aware fetch policies for smt processors. In *Proceedings of the 5th International Symposium on High Performance Computing*, pages 70–85. IEEE Computer Society, October 2003.

[22] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *34th Annual International Symposium on Microarchitecture*, December 2001.

[23] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[24] Compaq Computer Corp., Shrewsbury, MA. *Alpha 21264 Microprocessor Hardware Reference Manual*, February 2000.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algortihms*, chapter Single-Source Shortest Paths, pages 536–538. The MIT Press/McGraw-Hill Book Company, Cambridge, MA, 1990.

[26] J. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture(ISCA-27)*, 2000.

[27] Sun, IBM take processors multithreaded, multicore. *EE Times*, February 2003.

[28] Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Beng-Hong Lim, Mark S. Squillante, and C. Eric Wu. Evaluation of multithreaded processors and thread-switch policies. *International Symposium on High Performance Computing*, pages 75–90, 1997.

[29] J. S. Emer. Simultaneous multithreading: Multiplying alpha's performance. In *Microprocessor Forum*, October 1999.

[30] Joel Emer. EV8:the post-ultimate alpha. In *PACT Keynote Address (http://research.ac.upc.es/pact01/keynotes/emer.pdf)*, 2001.

[31] Keith I. Farkas, P. Chow, Norman P. Jouppi, and Z. Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *30th International Symposium on Microarchitecture*, December 1997.

[32] Brian A. Fields, Rastislav Bodík, and Mark Hill. Slack: Maximizing performance under technological constraints. In *To appear in the Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[33] Brian A. Fields, Rastislav Bodík, Mark Hill, and Chris J. Newburn. Interaction cost and shotgun profiling. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2004.

[34] Brian A. Fields, Shai Rubin, and Rastislav Bodík. Focusing processor priorities via critical-path prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[35] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, November 1995.

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

[37] Brian R. Fisk and R. Iris Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *IEEE International Conference on Computer Design*, Austin, TX, October 1999.

[38] D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2000.

[39] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[40] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th Annual International Symposium on Computer Architecture*, 1998.

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

[42] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ILP. In *12th International Conference on Supercomputing*, 1998.

[43] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–147. ACM Press, 2003.

[44] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[45] R Halstead and T Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. In *25th Annual International Symposium on Computer Architecture*, pages 443–451, 1998.

[46] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[47] T. H. Heil and James E. Smith. Selective dual path execution. Technical Report http://www.engr.wisc.edu/ece/faculty/smith_james.html, University of Wisconsin, Madison, November 1996.

[48] Seongmoo Heo, Ken Barr, and Krste Asanović. Reducing power density through activity migration. In *International Symposium on Low Power Electronics and Design*, August 2003.

[49] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[50] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[51] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *23rd Annual International Symposium on Computer Architecture*, pages 260–270, 1996.

[52] W. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy-efficiency and temperature management. In *MICRO33*, December 2000.

[53] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.

[54] IBM. Power5: Presentation at microprocessor forum, 2003.

[55] International Technology Roadmap for Semiconductors. 2003.

[56] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *25 Years ISCA: Retrospectives and Reprints*, pages 388–397, 1998.

[57] A. Kaufmann and G. Desbazeille. *The Critical Path Method*. Gordon and Breach, 1969.

[58] James E. Kelley. The construction scheduling problem (a progress report). Technical report, UNIVAC Applications Research Center Remington Rand UNIVAC, Nov 1957.

[59] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduling algorithm for ILP processing. In *International Conference on Parallel Processing*, 1996.

[60] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, December 1998.

[61] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, December 2003.

[62] Nam Sung Kim and Trevor Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *17th International Conference on Supercomputing*, June 2003.

[63] Artur Klauser, A. Paithankar, and Dirk Grunwald. Selective eager execution on the polypath architecture. In *25th Annual International Symposium on Computer Architecture*, page To appear, June 1998.

[64] John David Kubiatowicz. Closing the window of vulnerability in multiphase memory transactions: The alewife transaction store. Master's thesis, Massachusetts Institute of Technology, February 1993.

[65] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 81. IEEE Computer Society, 2003.

[66] Monica S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[67] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[68] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 128–137. ACM Press, 2003.

[69] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[70] Mikko H. Lipasti and John Paul Shen. The performance potential of value and dependence prediction. In *EUROPAR-97*, August 1997.

[71] Mikko H. Lipasti, C. B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[72] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreading processors. In *IEEE Transactions on Parallel and Distributed Systems, 10(9)*, September 1999.

[73] P. Geoffrey Lowney, Sefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.

[74] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[75] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technical Journal*, February 2002.

[76] R. L. Martino. *Critical Path Networks.* MDI Publications, 1967.

[77] Compaq chooses SMT for alpha. *Microprocessor Report*, 13(16), December 1999.

[78] Todd Mowry and Sherwyn Ramkissoon. Software-controlled multithreading using informing memory operations. In *Seventh International Symposium on High Performance Computer Architecture*, 2000.

[79] S. Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[80] Il Park, Michael Powell, and T. Vijaykumar. Reducing register ports for higher speed and lower energy. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[81] Y. N. Patt, W. M. Hwu, and M. Shebanow. Hps, a new microarchitecture: rationale and introduction. In *Proceedings of the 18th annual workshop on Microprogramming*, pages 103–108. ACM Press, 1985.

[82] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system, 1998.

[83] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, August 1998.

[84] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 54–65. IEEE Computer Society, 2001.

[85] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, N. H. Reilly, and M. J. Smith. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *Proceedings of the International Solid State Circuits Conference*, January 2002.

[86] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *2001 Workshop on Complexity-Effective Design*, June 2001.

[87] Ryan Rakvic, Bryan Black, Deepack Limaye, and John Paul Shen. Non-vital loads. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 165–174, February 2002.

[88] Joshua Redstone, Susan Eggers, and Henry Levy. Mini-threads: Increasing TLP on small-scale SMT processors. In *Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[89] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[90] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Second Annual ACM*

Symposium on Parallel Algorithms and Architectures, pages 169–178, July 1990.

[91] Toshinori Sato, Akihiro Chiyonobu, and Itsujiro Arita. Energy reduction via critical path prediction. In *Proceedings of the Workshop on Complexity-Effective Design held in conjunction with 29th International Symposium on Computer Architecture*, May 2002.

[92] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[93] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *28th International Symposium on Microarchitecture*, pages 57–68, Ann Arbor, MI, November 1995. IEEE.

[94] Manfred Schlett. Trends in embedded microprocessor design. *IEEE Computer*, 31(8):44–49, August 1998.

[95] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, page 29. IEEE Computer Society, 2002.

[96] John Seng. *Optimizing Processor Architectures for Power-Efficiency.* PhD thesis, University of California at San Diego, June 2003.

[97] John Seng, Eric Tune, and Dean Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[98] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth International Comference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2002)*, October 2002.

[99] Premkishore Shivakumar and Norm Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *Technical Report 2001/2, Compaq Computer Corporation*, August 2001.

[100] Burton Smith. The architecture of HEP. In *On Parallel MIMD computation: HEP supercomputer and its applications*, pages 41–55, 1985.

[101] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[102] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[103] A. Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[104] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *14th Annual International Symposium of Computer Architecture*, pages 27–31, June 1987.

[105] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *ISCA01*, June 2001.

[106] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Journal of Instruction Level Parallelism*, 1(1):1–24, 1999.

[107] Suphachai Sutanthavibul and Eugene Shragowitz. Dynamic prediction of critical paths and nets for constructive timing-driven placement. In *Proceedings of the 28th conference on ACM/IEEE design automation*, pages 632–635. ACM Press, 1991.

[108] Radhika Thekkath and Susan Eggers. The effectiveness of multiple hardware contexts. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[109] S. Thompson, P. Packan, and M. Bohr. MOS scaling: Transistor challenges for the 21st century. In *Intel Technology Journal*, Q3 1998.

[110] Jessica Tseng and Krste Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. In *In Proceedings of ISCA-30*, June 2003.

[111] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, December 2001.

[112] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[113] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[114] Dean M. Tullsen and Brad Calder. Computing along the critical path. Technical report, University of California, San Diego, October 1998.

[115] Dean M. Tullsen, Susan J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[116] Eric Tune, Rakesh Kumar, Dean Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th International Symposium on Microarchitecture.* IEEE, 2004.

[117] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, February 2001.

[118] Eric Tune, Dean Tullsen, and Brad Calder. Quantifying instruction criticality. In *Proceedings of the 11th Int'l Conference on Parallel Architectures and Compilation Techniques.* IEEE, 2002.

[119] Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest, and Joel Emer. Instruction fetching: Coping with code bloat. In *22nd Annual International Symposium on Computer Architecture*, pages 345–356, 1995.

[120] Carl Waldspurger and William Weihl. Register relocation: Flexible contexts for multithreading. In *20th Annual International Symposium on Computer Architecture*, 1993.

[121] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–188, April 1991.

[122] Steven Wallace, Brad Calder, and Dean M. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[123] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[124] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *27th International Symposium on Microarchitecture*, pages 1–11, San Jose, Ca, November 1994. IEEE.

[125] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. Technical Report CS-94-48, University of Utah, 1, 1994.

[126] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[127] C.-Q. Yang and B. P. Miller. Performance measurement for parallel and distributed programs: a structured and automatic approach. *IEEE Trans. Softw. Eng.*, 15(12):1615–1629, 1989.

[128] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[129] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In Doug DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, volume 27, 2 of *Computer Architecture News*, pages 42–53, New York, N.Y., May 1999. ACM Press.

[130] Victor V. Zhirnov, Ralph K. Cavin III, James A. Hutchby, and George I. Bourianoff. Limits to binary logic switch scaling: A gedanken model. *Proceedings Of The IEEE*, 91(11), November 2003.