Using Criticality to Attack Performance Bottlenecks

Brian Allen Fields

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-176
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-176.html

December 14, 2006

Acknowledgement

See inside document for the many acknowledgements.

Using Criticality
To Attack Performance Bottlenecks

by

Brian Allen Fields

B.S. (University of Cincinatti) 1999
M.S. (University of Wisconsin-Madison) 2001

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Rastislav Bodik, Chair
Professor John Wawrzynek
Professor Yale Braunstein

Fall 2006

The dissertation of Brian Allen Fields is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Using Criticality

To Attack Performance Bottlenecks

by

Brian Allen Fields

Abstract

Using Criticality

To Attack Performance Bottlenecks

by

Brian Allen Fields

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodik, Chair

We observe that the challenges software optimizers and microarchitects face every day boil down to a single problem: bottleneck analysis. A bottleneck is any event or resource that contributes to execution time, such as a critical cache miss or window stall. Tasks such as tuning processors for energy efficiency and finding the right loads to prefetch all require measuring the performance costs of bottlenecks.

In the past, simple event counts were enough to find the important bottlenecks. Today, the parallelism of modern processors makes such analysis much more difficult, rendering traditional performance counters less useful. If two microarchitectural events (such as a fetch stall and a cache miss) occur in the same cycle, which event should we blame for the cycle? What cost should we assign to each event?

In this work, we propose a new way of thinking about performance that employs a *global*

view of program execution to determine the *criticality* of program events. With the new-found understanding that a rigorous notion of criticality provides, we can correctly identify which events are bottlenecks, something that is not generally possible with event counts alone. Our work goes even further, however. We can also quantify how much a particular bottleneck *costs* to execution time, how much *slack* a non-bottleneck event has, as well as how the cost and slack of multiple events *interact* with each other.

This thesis makes the following key contributions:

- **Fundamental metrics.** We have gleaned from the seemingly diverse space of performance-related problems a few fundamental requirements of a performance analysis methodology for modern machines: measuring and interpreting $cost$, $slack$, and interactions.

- **Modeling.** We have shown how to construct dependence-graph models capable of representing the critical path of modern, parallel machines.

- **Measuring.** We have developed software algorithms for efficiently computing the $cost$, $slack$, and interactions for large numbers of micro-operations.

- **Interpreting.** We have characterized the types of interactions that are possible and shown how they can be exploited in design and optimizations.

- **Hardware Support.** We have developed inexpensive hardware capable of measuring criticality online and proposed a profiling infrastructure (enhancing performance counters) that enables measuring of the more sophisticated $cost$, $slack$, and interaction metrics for real program executing on real machines.

- **Case Studies.** We have illustrated through case studies how our performance analysis technology can tackle some of the problems that face architects and optimizers.

While this thesis focuses on microprocessors, many of the techniques are applicable to parallel systems in general. To illustrate the generality, we briefly illustrate how to model a chip multiprocessor using our dependence-graph abstraction. Once the graph is constructed, all of the criticality metrics discussed throughout the thesis can be measured and interpreted.

Professor Rastislav Bodik
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgments

The work described in this thesis is the result of the efforts of many people. In particular, I worked with Chris Newburn on the concept of interaction cost. We also worked together to make the shotgun profiler and criticality analyzer more palatable to the engineers that would have to build it. Shai Rubin was my early partner in the work during the initial formulations of the dependence-graph model. Renju Thomas and Mary Vernon have also worked on the project, providing new insights into what is and isn't possible with the dependence graph. I'd also like to thank John Kubiatowicz, John Wawrzynek and Yale Braunstein for serving on my committee and providing needed feedback for my thesis.

Intensive review and suggestions also helped improve our work. I would like to especially thank Amir Roth, Guri Sohi, and David Wood for their time, energy, and encouragement, especially during the heady times at the beginning of the project. I've also benefited substantially from conversations with Sarita Adve, Sanjay Patel, Mark Buxton, and the Intel Microarchitecture Research Lab (including Konrad Lai, John Shen, Ravi Rajwar, Srikanth Srinivasan, Jared Stark, and Chris Wilkerson).

I'd also like to thank the numerous reviewers of our work, which greatly improved communication of the ideas. Besides those that remain anonymous and those mentioned above, the following people have taken time to read less-than-polished drafts of our papers: Bradford Beckmann, Adam Butts, Jason Cantin, Pacia Harper, Alvy Lebeck, Jarrod Lewis, Dave Mandelin, Milo Martin, Paramjit Oberoi, Dan Sorin, Manu Sridharan, Min Xu, and Craig Zilles. I'm very grateful to all of them.

I started my PhD program in the computer architecture group at the University of Wisconsin-

Madison, before moving with my advisor to Berkeley midstream. The effect of attending Wisconsin is to be enveloped by a computer architecture mindset, and I am indebted to the people there for helping to develop my critical thinking abilities. The Architecture Industrial Affiliates Meetings were especially vitalizing to my research efforts.

Finally, my advisors, Rastislav Bodik and Mark Hill, have helped me mature into a professional. Both have taught me valuable research and (especially) communication skills that will be critical to my future career success. During the course of my PhD education, they were patient when I needed patience and expertly guided me towards independence by gradually reducing their role in the work. I was lucky to have stumbled across them early in my graduate career, before I was able to fully appreciate what being a good advisor entails.

Part I

First Part

Chapter 1

Introduction

During the design and optimization of a microprocessor, the quintessential question is "Where have all the cycles gone?" The answer is they've gone to *bottlenecks*, and designers will spend many months simply identifying the bottlenecks (*e.g.*, limited fetch bandwidth or a too small instruction window) of current processors in preparation for designing a new one. Furthermore, optimization systems (such as compilers or cache-line prefetchers) must judiciously choose which bottlenecks to target since most optimizations involve a difficult tradeoff between their benefit and the overhead they incur.

Despite its importance, bottleneck analysis has not kept pace with the increasing performance complexity of computer systems. The predominant form of analysis employed today, based on *event counts* (*e.g.*, number of cache misses), was invented at a time when processors were simple pipelines, which means they executed one instruction at a time and did so in program order. Today, even high latency events, such as cache misses, might have no effect on performance due to parallel execution with other instructions. In fact, we have found that the effective performance

cost of individual cache misses and branch mispredicts can differ by over an order of magnitude. This observation points to a limitation of event counts: What good is a cache-miss count if we don't know how costly those cache misses are?

In this work, we propose a new way of thinking about performance that employs a *global* view of program execution to determine the *criticality* of program events. By measuring criticality, we can correctly ascertain what events in a complex, parallel microprocessor are bottlenecks and which could be delayed without any performance harm.

1.1 Why Existing Analysis is Inadequate

Before describing our proposal, however, let's take a closer look at why existing performance analysis is inadequate for modern machines. The prevailing performance analysis is often ineffective at answering questions important to designers and programmers. The problem is that current analyses — designed for simple in-order pipelines — have been outgrown by complex architectures that exploit parallelism aggressively. To understand why, consider a typical equation used in Hennessey and Patterson's textbook [48] (Figure 5.9):

> CPU execution time = (CPU clock cycles + Memory stall cycles) × Clock cycle time
> Memory stall clock cycles = Number of misses × Miss penalty

where *miss penalty* is usually defined as miss latency.

This equation typifies *event-count analysis*, where the contribution of a processor resource (in this case, the memory hierarchy) is obtained by counting the number of events (*e.g.*, cache misses) and multiplying by the latency of the event (*e.g.*, miss penalty). While such equations work well for a simple in-order processor, they can be inaccurate for machines that exploit significant parallelism, *e.g.*, out-of-order or multiple-core processors.

Considering the simple example of a nonblocking cache causing two cache misses to execute completely (or partly) in parallel. The above equation would attribute two miss latencies to memory stall cycles, when, in reality, one latency was "free," because it was hidden under the other latency. With out-of-order execution, ALU operations can overlap with each other and with cache misses. In the latest version of the textbook, Hennessey and Patterson acknowledge the problem and lack of a good solution (p. 412).

Recognizing this problem, researchers over the past decade have often referred informally to concepts such as the "critical path" and "latency tolerance" to explain otherwise unexplainable effects on performance. Sometimes understanding these effects was so important for the research that ad-hoc methods were developed to quantify them. These methods lack an underlying appreciation for the nature of the problem, however.

For example, consider the commit-attribution breakdowns that are common in many papers [68, 79, 80, 107, 74, 48]. With this methodology, heuristics are used to blame particular dynamic instructions for cycles where no instructions are committed. Although this technique captures some parallel behavior that simple event counts miss (*e.g.*, our experiments showed the percentage of execution time attributed to memory stalls was mostly accurate), it is not able to explicitly measure the *interactions* between groups of events. An example of a typical interaction is ALU operations overlapped with cache misses; a more subtle one would be fetch stalls overlapped with issue bandwidth. This lack of accounting for interactions is evident in how the breakdowns are reported, with one category for each resource of interest (*e.g.*, cache misses, branch mispredictions). We know intuitively, however, that is impossible to account for 100% of execution time by assigning blame for each cycle to individual resources. Interactions cause multiple resources to be responsible

for a single cycle.

The Growing Importance of Bottleneck Analysis. We pointed out above why current performance analysis techniques often lead to inaccurate accounting in modern machines. What we have not discussed is why this problem is so important to solve. Our claim is that having a robust performance analysis methodology is much more important now than it was in the past and will become even more crucial in the systems of the future. The reason is that design and optimization tradeoffs that could be resolved through intuition in the past are now often too complex even for experts in the field. In the next section we will discuss some of the challenges that designers and optimizers face.

1.2 Bottleneck Analysis Applications

Processor Design. Performance effects often surprise designers, and the complexity increases each processor generation. For example, the effect of so-called "replay tornadoes" in the Pentium 4 — where a mispredicted data dependence of a load instruction causes cascading aborts of instructions that consume the incorrectly loaded data — can cause a great deal of performance variability in applications. Sometimes this sophisticated selective replay mechanism incurs penalties significantly greater than a simple "replay everything" approach. When this happens, the technique designed to eliminate a bottleneck becomes a bigger bottleneck than the one it is targeting.

Moving forward, the increasing performance complexity will be in the form of many-threaded execution. Intel's Hyperthreading [65], also known as simultaneous multithreading, has performance that is so variable that some users opt to disable it entirely. The bottlenecks are difficult to identify as they can involve complicated resource contention between multiple threads. In fact,

there is no way to know whether hyperthreading is helping or hurting except to disable it, run again the same set of applications with the same inputs, and see if that results in a speedup. If we could identify the bottlenecks, we could determine whether contention between threads or shared resources is what is hurting performance, enabling an intelligent decision as to whether to run one or more threads.

In the past, designers have often dealt with complexity in the same manner that civil engineers (used to) design bridges: over-engineer anything that is not understood well enough to be done precisely. For instance, if we suspect threads in an SMT machine are competing for scarce resources, we could simply make all resources abundant — or at least as many as chip space permits. This solution is no longer viable. The increasing importance of keeping energy consumption as low as possible necessitates judiciously choosing what work to do when. It has become very important to quantify bottlenecks, determining the performance effect of performing a piece of work now versus later.

To generalize the problem space, the processor design questions we address in this thesis include:

- **Performance Breakdowns.** How can we construct a complete breakdown of processor performance? For example, what percent of execution time should be attributed to cache misses or branch mispredicts; and what percent should be attributed to a combination of the two?

- **Design-space Search.** How can we search a huge design space quickly and with a high level of confidence that we did not settle for a local optimal? Understanding bottlenecks makes it easier to avoid building an unbalanced processor that could be improved by trading off, for example, an overly large instruction window for additional level-one cache.

Optimization in Hardware. An ever-present challenge in computer architecture is to make optimal use of scarce resources, whether those resources be limited functional units or a constrained power budget. Since some instructions and microprocessor events are bottlenecks while others are not, a priority system seems important for optimizing performance. This problem of resource arbitration is becoming more important since, although chip real estate is abundant, energy budgets are becoming tighter and increasing wire delays have introduced a new scarce resource of "spatial closeness". A related problem is knowing when to use resources to apply aggressive speculation and when such efforts could be better used elsewhere, *e.g.*, to run other threads.

In their quest to make most efficient use of available power, architects have explored ideas to reconfigure the hardware on the fly, both in terms of dynamically adjusting the frequency as well as resizing hardware structures, such as the instruction window (*e.g.*, [87], [8]). In performing such optimizations, they encounter a problem similar to that of hyperthreading above: it is very difficult to know whether an alternative configuration is effective without trying it out first. It is difficult to infer from performance counts whether a particular resource can be degraded without hurting performance and it is even more difficult to know whether increasing frequency or window size might improve performance. What is needed is a bottleneck analysis that can identify critical resources dynamically to guide reconfiguration.

In summary, some performance analysis hardware optimization questions designers struggle with include:

- **Resource Arbitration.** How should we effectively arbitrate for scarce resources? More specifically, what priority policies should be used for scheduling instructions and micro-operations to use limited cache ports, window slots, or functional units.

- **Speculation Control.** How should we control speculation to improve the risk/benefit trade-off?

- **Dynamic Reconfiguration.** How can we dynamically reconfigure the hardware and frequency to match the needs of a program?

Software Optimization. The process of optimizing programs presents a set of performance questions of its own. The traditional notion that a load that causes many cache misses should be prefetched, scheduled sooner, or otherwise given priority, is no longer necessarily true. As discussed above, instruction-level parallelism can "hide" the latency of some cache misses completely by performing other useful work during its processing, in which case the cache miss is no longer a bottleneck. In the future the problem will become much more daunting: it may be that an entire thread is hidden behind other threads, so that none of the cache misses in that thread are critical. Whether a thread is hidden will depend on many variables of the program's execution, such as the balance of work between processors, contention for shared memory, and the nature of data dependences between threads. A method for identifying whether individual cache misses (or other processor events) are bottlenecks or not will somehow need to consider all of these effects simultaneously.

In the near future, programmers will be presented with a new and, in many cases, extremely difficult problem: how to update their applications to take advantage of multiple processing cores. Of course, the multiple cores are only useful if they can reduce or eliminate the bottlenecks present in the single-threaded execution. As of now, the tools that support programmers in this task are woefully lacking. To determine how programs should be cut into task-sized chunks requires a very high level of expertise and judgment. Furthermore, there is no way to know whether one

configuration is better than another except to try them both out and see which one runs faster.

Software optimization problems can be summarized as:

- **Performance Profiling.** How can we know which instructions are bottlenecks (and should be targeted for optimization) and which should be left alone since they are already hidden behind other useful work?
- **Performance Estimation.** How can we save programmer effort by telling them in advance the effect of a given software organization (*e.g.,* the division of a program into tasks for execution on a CMP) before they rewrite the code?

1.3 Grand Challenges

Our primary thesis is that all of the above performance analysis questions can be addressed with a fundamental understanding of the *criticality* of events during a program's execution. When we use the word "event" here, we are assuming a broad definition which encompasses all sorts of architectural and microarchitectural features, such as cache misses, branch mispredicts, load-store-queue-full stalls, and multiprocessor communication. We will explore the notion of criticality by posing three challenge questions that have directed the course of our work.

- **Cost.** How *costly* is an event to execution time? (*i.e.,* how much can execution time possibly be reduced by optimizing the event?)
- **Slack.** How much *slack* does an event have? (*i.e.,* how much can the event be slowed down or a resource reduced in size before increasing execution time?)
- **Interactions.** How do the cost and slack of multiple events *interact* with each other?

It is clear how answering these questions would lead to solutions to some of the performance analysis problems discussed in the previous section. For example, knowing how costly each instruction execution is would make designing an effective *resource arbitration* policy straightforward, since the most costly should be given highest priority for scarce resources. For other problems, the connection may not be immediately obvious, *i.e.*, how can we use cost, slack and interactions to enable a quicker *design-space search*. Part of our work in developing our thesis will be to illustrate the connection through case studies (Chapter 7).

1.4 Our Approach

We set out to tackle the above challenges in three steps. The first is to develop a new way of modeling program performance that is complete enough for us to measure criticality metrics, since it is clear that traditional performance counters are not sufficient (Chapter 3). The second step is to develop algorithms for measuring and interpreting cost, slack, and interactions (Chapter 4). Finally, to make the methodology usable in practical optimizations on real software, we proposed hardware support for measuring criticality metrics online as well as a new profiling infrastructure to enhance existing performance counters (Chapter 6).

Modeling. A crucial requirement to achieve our goals is to capture the parallelism in the program execution. The most straightforward way to do this would be to observe the entire set of events that occur each machine cycle. At first glance, such a representation would appear to encompass all you would ever need to know about a program's execution. It misses a critical component, however, that is needed to understand *why* each event occurred when it did (as opposed to earlier or later). For this purpose, we need to observe the *dependence constraints* that occur between the events. In

fact, a graph containing these dependence constraints, labeled with latencies, is all that is needed to effectively represent a program execution's performance behavior.

In the dependence graph, each node represents the beginning of a very low-level micro-operation — examples include an instruction being fetched and an instruction being "woke-up" once it becomes data ready. Edges between these nodes represent constraints that must be satisfied for each micro-operation to "fire". For example, if the machine requires in-order fetching of instructions, dynamic instruction $i + 1$ cannot be fetched until after i is fetched. Thus, an edge would be placed from i's fetch node to $i + 1$'s fetch node. If i happened to be a branch that was mispredicted, an edge would be placed from i's execute node to $i + 1$'s fetch node, since the correct output of the branch must be produced before the next correct-path instruction can be fetched. A complete examination of how the graph models are constructed and validated for a particular machine is the subject of Chapter 3.

Measuring and Interpreting. Relatively simple algorithms can be applied to these dependence graphs to compute *cost* and *slack*. Measuring and especially interpreting *interactions* requires some additional theoretical grounding, however. If two micro-operations a and b interact, it means that optimization decisions involving cost or slack should consider both a and b together. In other words, if we make optimization decisions targeting a, those decisions might impact the cost or slack characteristics of b.

We have found it most practical to treat interactions differently when doing a $slack$ analysis than when performing a $cost$ analysis. The quintessential example of an interaction involving $cost$ is two cache misses that are being serviced simultaneously. If we consider either miss in isolation, it may appear there would be no benefit in performing an optimization, since the other miss

would still limit performance. In other words, the individual $cost$ of either miss is zero cycles. If we optimize both misses, however, substantial performance improvement could result. In other words, the *aggregate cost* of the two misses together is large. We refer to this type of interaction as a *parallel interaction*, since it occurs when multiple events occur in parallel.

There is no equivalent to a parallel interaction for $slack$. Instead, slackful events often exhibit what we call *serial interactions*, which, as the name implies, occurs between micro-operations in series — *i.e.*, when there is a dependence-edge chain leading from one micro-operation to the other. The interaction occurs when an entire (non-critical) dependence-chain has some number of $slack$ cycles. Since increasing the latency of one micro-operation increases the latency of the entire chain, the micro-operations on the chain must *share* the slack cycles when performing optimizations. For example, if eight cycles of slack are available on the chain, the sum of the increased latency of all micro-operations on that chain cannot exceed eight without increasing execution time. We deal with these sorts of interactions by *apportioning* a specific share of the slack to each micro-operation prior to performing optimizations. Since the amount a micro-operation can be delayed is bound by the amount it is apportioned, the apportioning policy is very important for optimization success.

During the course of the research we were surprised to discover that serial interactions exist for $cost$ as well. Two micro-operations on the critical path (a and b) will often exhibit a serial interaction when there is a nearly critical secondary path. In the case of a serial interaction, a and b share a certain number of $cost$ cycles, meaning that improving either a or b can eliminate those cycles.

In summary, there are two ways of dealing with interactions: (1) measure explicitly the

degree of interaction between each pair (or triple, etc.) of events or (2) use an apportioning scheme that takes interactions into account but, after apportioning is complete, the optimization engine does not need to have any knowledge of them. We have found in practice that the first scheme is most suitable for $cost$ analysis while the second is best for $slack$ analysis. A complete discussion of $cost$, $slack$, and interactions is the subject of Chapter 4.

Hardware Support. While the graph model can be used to efficiently compute criticality metrics using standard algorithms in software, hardware presents much more restrictive constraints on the implementation. Due to these constraints, some innovation is required to effectively compute $cost$, $slack$, and interactions for real programs running on real machines.

We take two basic approaches to developing hardware solutions depending upon how the criticality measurements will be used. If we plan to use the measurements for "tight-loop" dynamic optimization, where the criticality information is consumed immediately by a control policy, a pure hardware solution is required. On the other hand, if we wish to understand the performance characteristics of an application in order to reorganize or optimize the software code, a better profiling system analogous to currently existing hardware counters may be all that is needed.

For the pure hardware approach, we developed a criticality analyzer that can efficiently detect with high accuracy whether a single dynamic micro-operation is on the critical path or not. The analyzer is very lightweight, requiring two components. The first is a set of hardware "probes" that detect *last-arriving edges* going into each node of the graph model. A last-arriving edge is the dependence constraint that delays the beginning of its target micro-operation the longest. Detecting last-arriving edges is relatively easy for the hardware since, during the actual execution, it is usually easy to simply observe which dependence constraint is resolved last. The second component is a

mechanism for *token-passing*, described below.

The algorithm for the criticality analyzer works by planting a token into a node n and then passing that token forward along last-arriving edges (copying the token if a node has multiple outgoing last-arriving edges). The token-passing terminates if all copies of the token reach nodes with no outgoing last-arriving edges. Since critical edges are necessarily last-arriving edges, n is definitely not on the critical path if the token-passing terminates. The longer the token-passing proceeds without terminating, the more likely n is on the critical path. Token-passing can be implemented in one of two ways: by a read-modify-write sequence on a small array or inline within the core of the machine. We discuss the tradeoffs of the two approaches in Chapter 6.

While the advantages of the criticality analyzer are that it is inexpensive to implement and provides quick feedback for optimizations, the disadvantage is that it only detects simple criticality and not $cost$, $slack$, or interactions. We developed a simple enhancement to detect $slack$, albeit with a substantial reduction in accuracy, but for the more sophisticated metrics a different approach is required.

For measuring $cost$, $slack$ and interactions, we developed a new profiling system suitable for replacing performance counters. The profiling system works by collecting a small amount of information from the hardware executing the program such that segments of the dependence graph for that program can be constructed offline. The hardware is kept inexpensive by allowing the sampling to be sparse: only one dynamic instruction needs to be monitored by the hardware at any one time. Dependence graph segments can still be reconstructed from these samples since, empirically, the same sequence of dynamic instructions (with similar microarchitectural behavior) recurs often in a program run. It is the most frequently recurring sequences for which it is most

important to construct a representative graph. Samples of individual dynamic instructions are glued together to form a sequence by matching a small number of *microarchitectural context bits*. These context bits are collected for each sample by the hardware and consist of one bit indications of whether cache misses, branch mispredicts, etc., occurred in the vicinity. Once the graph fragments are constructed offline, all of the standard algorithms for computing cost, slack and interactions can be applied. We call the system *shotgun profiling*, due to the similarity of the algorithm to shotgun genome sequencing [40]. A complete discussion can be found in Chapter 6.

Summary. The goal of our work is to provide a new set of analysis tools to tackle the difficult performance problems facing architects and optimizers targeting the complex, parallel computer systems of the future. To this end, we have made the following key contributions:

- **Fundamental metrics.** We have gleaned from the seemingly diverse space of performance-related problems a few fundamental requirements of a performance analysis methodology for modern machines: measuring and interpreting $cost$, $slack$, and interactions.

- **Modeling.** We have shown how to construct dependence-graph models capable of representing the critical path of modern, parallel machines (Chapter 3).

- **Measuring.** We have developed software algorithms for efficiently computing the $cost$, $slack$, and interactions for large numbers of micro-operations (Chapter 4).

- **Interpreting.** We have characterized the types of interactions that are possible and shown how they can be exploited in design and optimizations (Chapter 4).

- **Hardware Support.** We have developed inexpensive hardware capable of measuring criticality online and proposed a profiling infrastructure (enhancing performance counters) that

enables measuring of the more sophisticated $cost$, $slack$, and interaction metrics for real program executing on real machines (Chapter 6).

- **Case Studies.** We have illustrated through case studies how our performance analysis technology can tackle some of the problems that face architects and optimizers (Chapter 7).

Considering that it appears no longer possible to simply expect better transistor technology to provide increased value for new machines, there will be increased pressure on designers to produce innovative solutions, for which understanding performance is crucial. We believe our contributions have substantially improved the state-of-the-art of performance analysis, providing methodologies that should help architects and optimizers deal with the high level of parallelism in the increasing complex machines being built today.

Chapter 2

Related Work

The use of critical path and slack in project management was introduced by the Navy in 1958 as part of the Polaris project, whose goal was to develop a system capable of launching ballistic missiles from submarines [115]. The Program Evaluation and Review Technique (PERT) charts developed for this project had a node for each “milestone” and each incoming edge to a node was a task that needed to be completed before the milestone was reached. This graph construction has become very popular in managing large projects and bares some similarities to the modeling that we introduce for microprocessors in Chapter 3.2.1.

One of the first works that considered parallelism in microprocessors when determining the performance effect of events (specifically, of cache misses) was by Srinivasan and Lebeck [104]. They defined an alternative measure of the critical path, called latency tolerance, that provided non-trivial insights into the performance characteristics of the memory system. Their methodology illustrated how difficult it is to measure criticality even in a simulator wherein a complete execution trace is available. Their latency tolerance analysis involves rolling back the execution, artificially

increasing the latency of a suspected non-critical load instruction, re-executing the program, and observing the impact of the increased latency. While their methodology yields an interesting analysis of memory accesses, their analysis cannot (easily) identify criticality of a broad class of microarchitectural resources, something that we can achieve with our modeling. A rollback simulator also presents some practical problems to use a measurement tool since it will tend to be a very difficult piece of software to write and is unlikely to be highly efficient in his measurements.

Heuristic Predictors of Criticality. Once it became clear to the architecture community that it was important to take parallelism into consideration when developing policies for optimizations, many efforts were made to estimate which events were critical and which were not. Without a criticality model to guide this estimation, researchers resorted to various heuristics. While these heuristics are sometimes successful in various circumstances, we have found through experimentation that they do not provide the high-quality criticality identification over a large range of microarchitectural events that our modeling achieves. A few of these heuristics are explored alongside our criticality work in later chapters.

One of the first heuristic predictors was proposed by Fisk and Bahar [39]. They explored hardware approximations of Srinivasan and Lebeck's latency-tolerance analysis. Their first heuristic used absolute performance as its indicator: if IPC degrades below a certain threshold while a miss is being serviced, that load is considered critical. This is very similar to the approach used in many hardware structure resizing papers (*e.g.*, [41, 8]). They also looked at heuristics based on the number of dependencies that exist on a cache-missed load's dependence graph. The intuition here is that, if a load's data is going to be used by many instructions, perhaps it is important to performance (hence critical).

Calder *et al.* [19] guide value prediction by identifying the longest dependence chain in the instruction window, as an approximation of the critical path, without proposing a hardware implementation. This approximation loses accuracy for two reasons: (1) they are only examining a very small, local region of the execution, while criticality is a global characteristic and (2) they only look at data-dependence chains even though resource constraints are often a more influential factor in determining the critical path. These two inaccuracies are related since, as we show in Chapter 3, it is necessary to model resource constraints in order to obtain a global measure of criticality.

Tune *et al.* [110] presented a systematic study of a large number of heuristics-based predictors, using the notion of the critical path as intuition. These heuristics looked for various architectural events that could indicate an instruction was critical. For instance, one of their heuristics identified any instruction that reaches the head of the re-order buffer before being executed as critical. We have found in our experiments that their heuristics often miss some critical instructions and, even more significantly, falsely identify a large number of noncritical instructions as critical.

Tune *et al.* also developed a methodology for judging the accuracy of a critical-path predictor, which involved measuring the ratio of the performance improvement from reducing the latency of instructions identified as critical to those identified as noncritical — the higher the ratio, the better the predictor. We use their methodology as part of our validation procedure (Section 3.4).

Srinivasan, *et al.* [103] proposed a heuristics-based predictor of load criticality inspired by their above mentioned latency-tolerance analysis. Their techniques consider a load as critical if (a) it feeds a mispredicted branch or another load that cache misses or (b) the number of independent instructions issued soon following the load is below a threshold. The authors perform experiments with critical-load victim caches and prefetching mechanisms, as well as measurements of the critical

data working set. Their results suggest criticality-based techniques should not be used if they violate data locality. The authors suggest there may be other ways for criticality to co-exist with locality. For example the critical-path could be used to schedule memory accesses.

Slack Measurement. The concept of latency tolerance explored by Srinivasan, *et al.* is directly related to slack. If the machine can "tolerate" more cycles of latency than a cache miss exhibits, that cache miss has an amount of slack equal to the difference. A couple of works preceded our work on slack.

Casmira and Grunwald [20] discussed the use of slack for power saving in a machine with multiple-speed clusters. The notion of slack used here, however, is limited to the number of cycles an instruction's execution can be delayed without delaying *any* instructions immediately consuming its data. Our experiments have shown exploiting only this *local* notion of slack leaves a lot of opportunity untapped (see Chapter 4).

Following our initial critical-path modeling work, Semeraro, et al. [95] used a dependence-graph model similar to ours for doing an offline slack analysis to determine when different parts of the machine can be executed at a slower rate, for power efficiency. Their notion of slack is global in nature and very similar to ours. Their exploration was restricted to how slack could be used for this particular application, however.

Cost Measurement. The first attempt to estimate the costliness of different classes of events in a microprocessor were event counters [5, 119]. These counter metrics have become standard and, before superscalar, out-of-order processors, was all that was needed. As discussed earlier, however, parallelism complicates analysis substantially.

In response to the problems with counters, ProfileMe [28] supports pair-wise sampling, where the latencies and events of two simultaneously in-flight instructions are recorded. With pair-wise samples, one can determine the degree that two instructions' latencies overlap in time. As far as existing processors are concerned, the Pentium 4 [26, 102] has a limited ability to account for overlapping cache misses. The most significant limitation of these profiling infrastructures is a lack of a methodology to interpret the results in a meaningful way. For example, it is not clear how to use the collected data to compute a complete breakdown of execution time. We propose our own profiling infrastructure that was designed with the specific goal of measuring criticality metrics in mind (Chapter 6).

Many researchers have realized the limitation with counters in computing performance breakdowns, but have still found a need for understanding where execution time is going in their proposed systems. Their solution is to *attribute* execution cycles to various culprits based on heuristics. For example, *commit attribution* [68, 79, 80, 107, 74, 48] assigns the blame for an unused commit cycle to the (uncompleted) instruction at the head of the reorder buffer during that cycle. Similarly, *fetch attribution* [31, 70] assigns blame for a wasted fetch cycle to the next instruction to be fetched. One work, by Sasanka, *et al.* [87], combined the attribution with some data-dependence information to increase accuracy. We have found empirically that these analyses do, indeed, accurately estimate the cost of certain classes of events (*e.g.*, data-cache misses), which was their intended purpose. Their generality is limited, however, in that the costs of many classes of events cannot be accurately computed (*e.g.*, fetch bandwidth) nor are they able to compute interaction costs at all.

Following our criticality and slack works, Tune et al. [111] used our dependence graph to

compute the cost of *individual* instructions in a simulator. We employ an algorithm very similar to theirs. The work did not consider interactions, however.

Miller, *et al.* [66] and Hollingsworth, *et al.* [51] use a metric called *critical-path zeroing* in the context of parallel software that is similar to our cost metric. They create a program dependence graph, taking into consideration inter-thread communication, to identify the critical path through the program. To identify the execution time that should be attributed to specific procedures, they zero out the edges that represent that procedure and compute the difference between the new critical path and the old. Our cost metric differs in that edges are not zeroed but instead *idealized*. The notion of idealization is specific to the individual microarchitecture event(s) that are being measured.

At a higher level of abstraction, the MACS model of Boyd and Davidson [14] assigns blame for performance problems to one of four factors: the machine, application, compiler-generated code, or compiler scheduling. They accomplish this by idealizing one factor at a time (to determine its cost). In comparison to this work, we do not examine high-level compiler decisions and focus on performance analysis at a more fine-grain scale. The MACS work also does not propose a way to measure interactions.

Interactions. We have found in our work that measuring the cost of an event or set of events is not very useful in a highly parallel machine, unless you also consider interactions between events. The reason is that there is often more than one event that is responsible for a certain amount of execution time.

Standard allocation of variation techniques do provide a way to quantify these interactions [54]. The techniques are inadequate for our purposes, however, for two reasons. First, for mathematical reasons, the effects are squared, but this squaring reduces interpretability, especially

when constructing a complete breakdown of performance. Secondly, no distinction is made between positive and negative (parallel and serial) interactions. As we show in our work, this distinction is very important for performance understanding.

Following our cost and interaction work, Karkhanis and Smith proposed an analytical model for out-of-order superscalar processors [57]. The primary advantages of their model are its simplicity and ability to provide quick insights by evaluating analytical equations as opposed to re-simulating (or performing a graph analysis.) Its disadvantages include a specificity to out-of-order superscalar processors and incomplete accounting for interactions (they only consider parallel, not serial, interactions).

Note that Karkhanis and Smith confirm empirically that in the microarchitecture they study the interactions (called “overlaps” in their paper) of branch mispredicts and icache misses with dcache misses are relatively insignificant (in other words, that the resources are nearly independent.) This discovery of near independence permits them to ignore interactions with a low, bounded error. For other resource classes or microarchitectures, interactions may be much more significant, as illustrated by the case studies in this thesis.

Applications of Criticality. Work related to our case studies will be discussed alongside each case study in Chapter 7.

Chapter 3

Microexecution Dependence Graphs

In the informal jargon of computer architecture literature, the preposition “on the critical path” is used to describe an event that causes other events to stall, waiting for its completion. For instance, a cache miss that is fetching a result needed by many instructions that come later would be considered a “critical” cache miss. This informal notion, however, is only a local characteristic of performance. The fact that an ALU operation had to wait for a cache miss to complete does not necessarily imply the cache miss is critical, since it may not have been important that the ALU operation be processed quickly.

The problem we tackle in this chapter is to understand what constitutes the critical path in a modern processor exhibiting substantial parallelism. To this end, we will present a methodology for constructing a *graph model* that represents the performance of a program executing on a specific processor. From this graph model it will be easy to determine the program execution’s critical path.

The first step is to derive the requirements for such a model. Specifically, we seek to understand what are the necessary components that the model must maintain in order to accurately

represent the critical path of a program execution.

3.1 Requirements

To begin our requirements specification, we will consider the essential characteristics of the critical path. The most basic is **performance sensitivity.** Performance sensitivity implies two properties:

- **For critical events.** If the latency of an event on the critical path is increased, then the execution time of the program should also increase. If the execution time stays the same, then clearly the event is not critical.
- **For noncritical events.** In contrast, if an event is not on the critical path, decreasing it's latency should not have any effect on execution time at all.

The implications of this requirement guide the rest of our work on criticality. As far as the critical path in particular, we can logically deduce the important properties that it must maintain. If we are to meet the requirement above that increasing the latency of a critical-path event e increases the latency of the entire program, it is logically true that e's increased latency delays the completion of the last instruction in the program.

In other words, starting from any critical event, there must be a sequence of *dependent events* leading all the way to the end of the program. One event e_2 is dependent upon another e_1 if e_1 needs to have occurred before e_2 can occur. Furthermore, since delaying the start of a program obviously delays the completion of the program (the program's first event is critical), we can conclude that the critical path must be a *sequence of dependent events that spans from the*

beginning of the program all the way to the end.

3.1.1 Implications for a Processor's Critical Path

Supporting a rigorous notion of the critical path requires rethinking the way performance is measured by current profiling and compilation infrastructures. In particular, the above requirements suggest the model should not be centered on event counts but rather on *dependences*, which is similar in nature to the way compilers compute the critical path through data-dependence graphs.

The significant difference is that *data* dependences alone are not enough to capture the critical path of a program executing on a microprocessor. This claim should be intuitive to architects: it's not just data dependences that affect microarchitectural performance. Other factors such as instruction window stalls, branch mispredicts, and functional-unit contention are also important. In fact, these **resource constraints** often have a bigger influence on overall performance than data dependences. Certainly if these factors affect performance they will be part of the program execution's critical path. Thus, to generalize, the model must take into account a processor's resource constraints (such as stalls and mispredicts) as well as data dependences.

The familiar model that incorporates all of these constraints is a cycle-accurate simulator. The limitation of a simulator, however, is that it only provides the critical-path *length* (execution time) and not its *composition*. There has been some work that attempts to enhance simulators with "roll-back" capability, which can provide some limited information about critical-path composition [104]. The roll-back approach is very cumbersome to implement however, and the resulting solution is inefficient to the point of being impractical for many applications.

Our solution is to start with data-dependence graphs, for which it is possible to determine critical-path composition, and add resource constraints to them. We want all of the constraints to

be incorporated in a *uniform* way, so that the graphs can be analyzed using standard approaches, as opposed to requiring caveats and special conditions for each of the various types of resources. It is not clear how to produce a uniform model, however, given that processors have such a great diversity of types of resource constraints that affect performance. How can such varied events such as branch mispredictions, finite fetch bandwidth, and re-order buffer stalls be included in the same model?

Definition of Microexecution. Before discussing how to design a model, we need to setup the problem a bit more precisely. The model will be a set of rules to construct dependence graphs representing the performance of *dynamic execution traces*, where a trace is a sequence of dynamic instructions. This is in contrast to a graph of static instructions, typical of those used by compiler writers. Our model will be, in this way, more similar to a simulator, which operates on the dynamic sequence of instructions. The significant distinction from a simulator, however, is that we will assume information is available concerning the *microexecution* of the program running on the processor. We define microexecution to be the microarchitectural characteristics of a program's execution (*e.g.*, branch mispredicts, stalls, and cache misses) as well as its normal functional behavior. These microarchitectural characteristics will be used to form some of the rules of the model.

3.2 Constructing a Graph from a Program Execution

There is an important distinction between a model of a microexecution and a model of a processor implementation. The latter can accurately measure performance (at least the critical-path length) even if some aspect of the execution is *altered*, *e.g.*, changing the latency of a cache miss. (These types of alterations will be central to our more advanced criticality analyses.) Microexe-

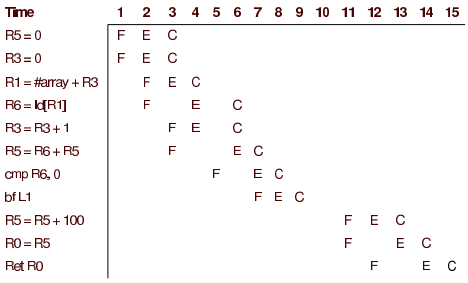
cution models, however, only represent the performance of a *particular execution*. We will begin the discussion with a model for constructing graphs of microexecutions (Section 3.2.1) and then continue to the more difficult task of constructing graphs capable of representing the performance effects of alterations (Section 3.2.2).

3.2.1 Graph of a Microexecution

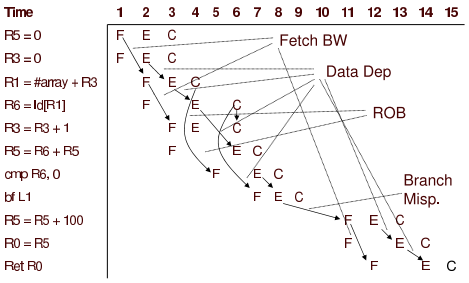
The traditional way to trace the microexecution of a complex pipeline is to use what is known as a "waterfall" timeline, such as the one shown in Figure 3.1(a) for a three-stage out-of-order pipeline. Here, each dynamic instruction occupies a row of a table while each column represents a machine cycle. The benefit of using a diagram such as this is that it visually illustrates the machine cycle that each micro-operation occurs. This makes it easier to see the performance effect of a stall due to a particular resource constraint, since certain micro-operations are delayed. Such diagrams have limited practical use, due to their layout, but they are often used for instructive purposes.

As a step towards creating a model of the microexecution, we make the dependence between resource constraints and micro-operation constraints in the waterfall diagram explicit, by adding edges as in Figure 3.1(b). Now we have a graph where any particular micro-operation cannot fire until after all the constraints represented by its incoming edges are resolved.

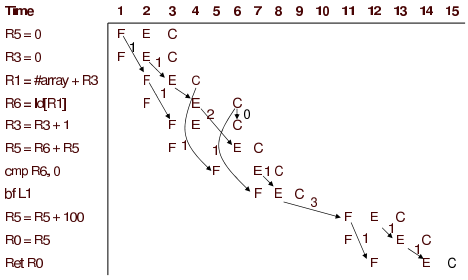
As it is, however, the graph still uses the timeline to account for the latencies of the various operations. By labeling each edge with the appropriate latency (Figure 3.1(c)), all of the information is included in the graph without the need of a timeline (Figure 3.1(d)). With the graph representation of the microexecution, finding the *critical path* is trivial: it is the longest path of dependences through the graph (the bold dependences in Figure 3.1(d)). As we will show throughout the thesis, the graph also enables many automatic analyses employing standard graph algorithms.



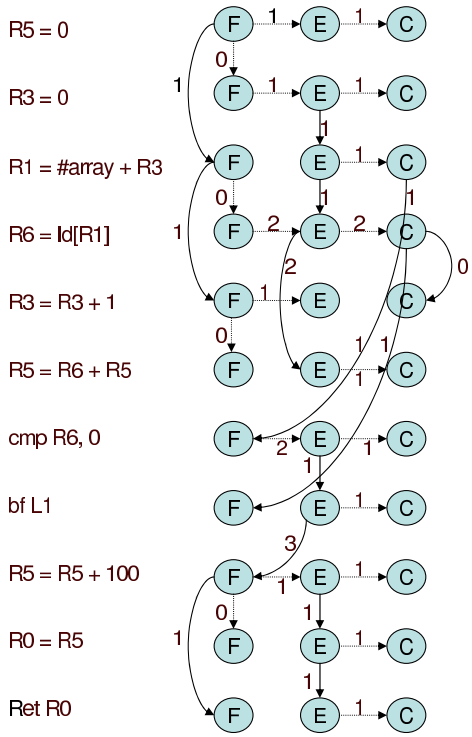
(a) Pipeline waterfall diagram.



(b) Waterfall diagram with edges.



(c) Edges labeled with latencies.



(d) Directed acyclic graph (DAG).

Figure 3.1: **Waterfall diagrams.** Traditionally, "waterfall" timeline diagrams, such as pictured in **(a)**, have been used to describe and analyze the performance of a microprocessor pipeline. We illustrate here a very simple 2-way superscalar out-of-order processor consisting of three pipeline stages (**F**etch, **E**xecute, and **C**ommit) and an instruction window with *four* entries. If we add edges to the diagram to indicate the dependences that result in stalls (as in **b**) and then label those edges with the latencies of the operations causing the stalls (as in **c**), we can convert the waterfall representation to one of a directed acycle graph (as in **d**). Notice how the graph representation retains all of the information in the waterfall diagram but in a more manageable way for computer-based analysis.

Notice the power and generality of this transformation: any microexecution that can be represented by a waterfall diagram can be represented as a dependence graph (and its critical path can be identified.) Thus the long-standing problem of identifying the critical path of a program executing on a complex processor is now possible for effectively any microarchitectural design.

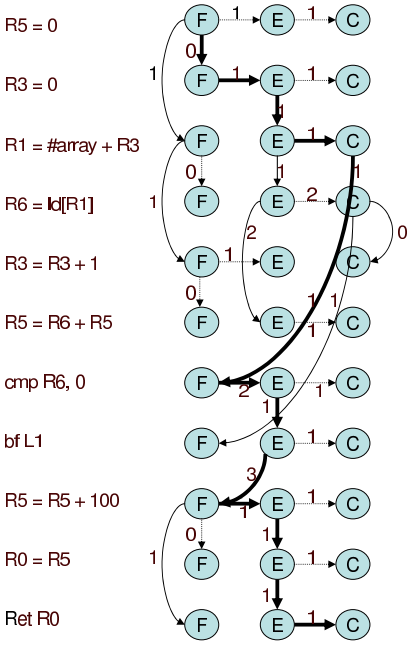
3.2.2 Complete Dependence Graph

We will now discuss the more difficult problem of constructing a model that not only captures the performance characteristics of the microexecution, but can also measure the effect of alterations to the program's execution. As a simple example of why this would be necessary, consider the problem of measuring the performance *cost* of a cache miss. We define the cost of the miss as the number of cycles of execution time that can be attributed to it. In other words, we need to know the performance of the execution altered such that the miss is turned into a hit.

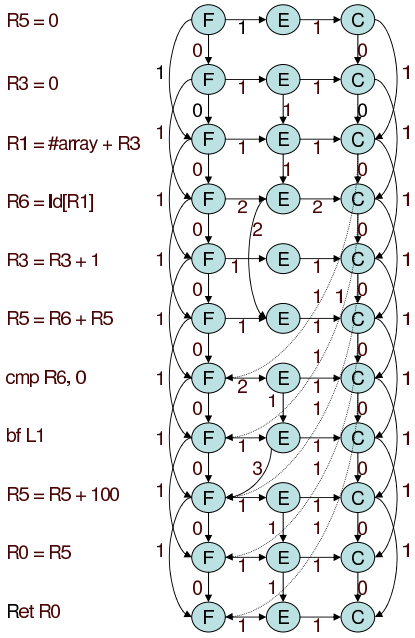
Another way of thinking about the problem is that it is not enough to simply know that the miss is on the critical path. Instead, we need information about the second-most critical path — specifically, how close it is to the critical path. For even more sophisticated analyses (discussed in later chapters), we might need to examine the third- or fourth-most critical paths.

Our solution is to construct a graph that contains all of these "secondary" paths. To do this, we need to model those dependence constraints that exist in the machine which are not the foremost limiters of performance. These dependencies do not affect the execution time for the current microexecution — since other dependencies delay the affected micro-operations longer anyway — but they may constitute portions of secondary critical paths.

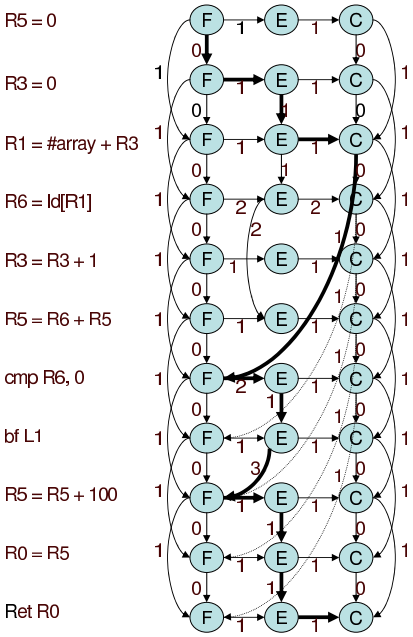
As an example of such a dependence, consider the code snippet of Figure 3.2(a). The 2-way machine's fetch bandwidth constraint inhibits the *cmp R6, 0* instruction from being fetched



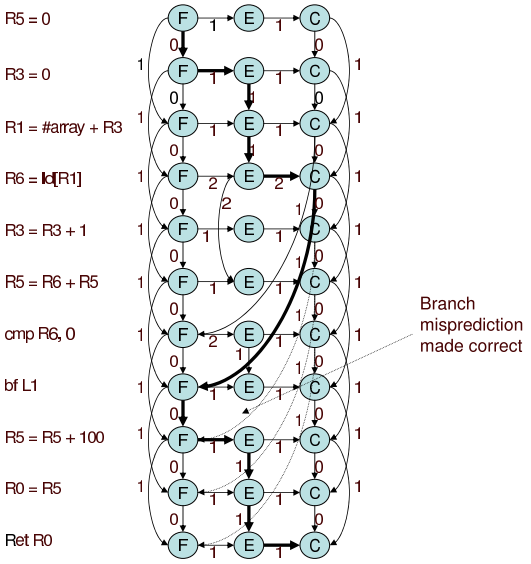
(a) Dependence graph with critical path.



(b) Complete dependence graph.



(c) Complete dependence graph with critical path.



(d) Dependence graph of altered microexecution.

Figure 3.2: **Dependence-graph diagrams of the microexecution.** Once a dependence graph is constructed, it is easy for a computer to identify the critical path through the microexecution using a topological sort (as in **(a)**). For more sophisticated analysis that makes use of "secondary" critical path information, a more complete dependence graph is required (see **(b)**). In this graph, edges are included for dependences that exist in the program execution even if they do not affect the performance of this particular microexecution. One example of the sophisticated analysis includes determining the *cost* of a branch misprediction. This quantity can be found by comparing the critical-path length of the graph with the misprediction to the critical-path length of the graph without it. The critical path of the altered execution is shown in **(d)**.

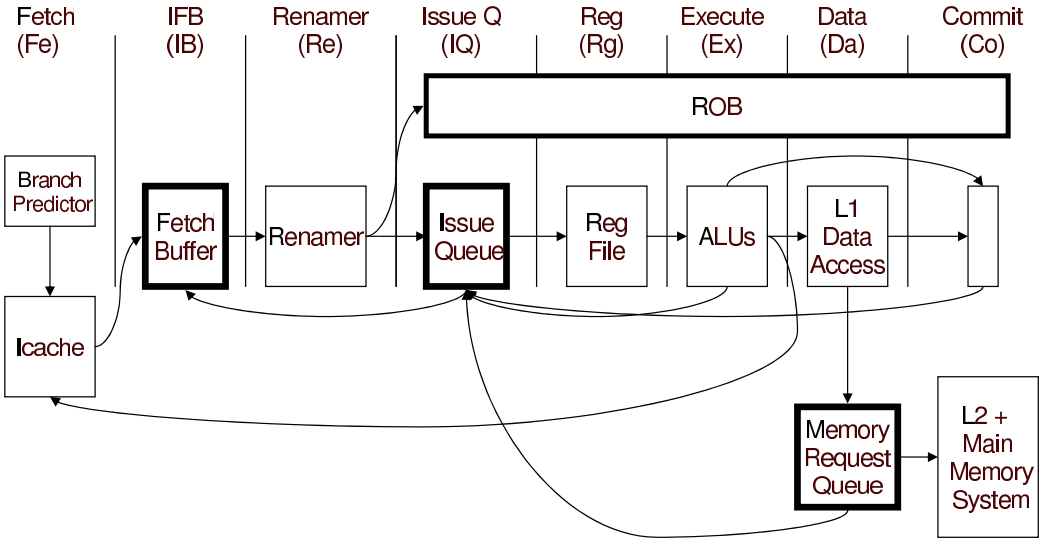
until after *R3 = R3 + 1*. The edge does not exist in the microexecution graph, however, because the fetch bandwidth is not saturated at that point in the execution (instead the fetch of *cmp R6, 0* is delayed by an instruction window limitation). In order to model secondary critical paths, all such dependences must be modeled.

Figure 3.2(b) illustrates a complete graph, including all important dependence constraints, for our example machine. Of course, the critical path is still easy to identify via a topological sort (Figure 3.2(c)). We can also now answer questions about performance costs. For example, the cost of the highlighted branch misprediction is equal to the difference in critical path lengths for the graphs in Figures 3.2(c) and 3.2(d).

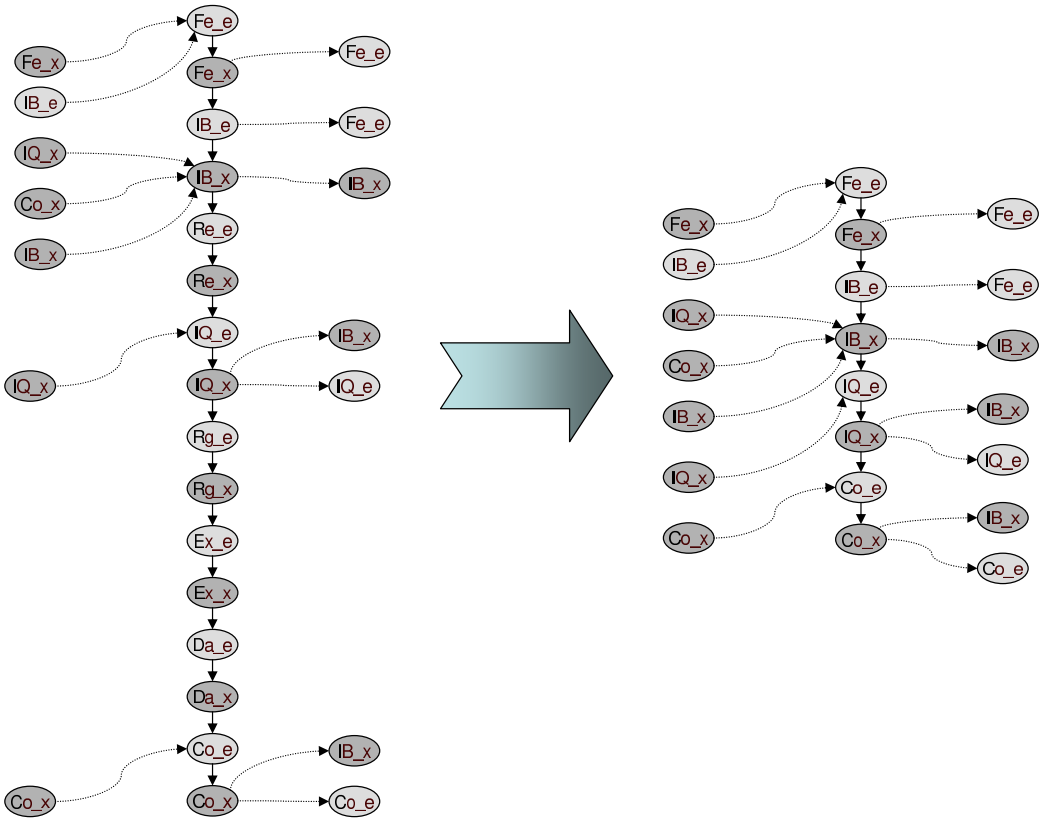
While constructing graphs of microexecutions is a straightforward, even automatable task, constructing full dependence graphs requires very detailed expertise of the resource constraints a machine imposes on an instruction stream. In the next section, we present a procedure that should make the task easier, by using information architects designing a new machine would have available anyway.

3.3 Procedure For Graph Model Development

For the simplified example processor above, we derived a dependence graph from a waterfall diagram and then added edges for other machine dependences that did not happen to cause stalls in that particular microexecution. While this procedure is good for explanation and useful for constructing a graph from a particular program execution trace, automating graph construction as well as understanding a processor's performance is made easier by developing a general graph *model* of the machine, from which a graph can be derived for any program's execution. In this section, we



(a) Typical superscalar pipeline.



(b) Node per-pipeline stage graph model.

Figure 3.3: **Converting pipeline to graph model and reducing.** A graph model can be derived directly from a pipeline model of the processor by (1) creating entry and exit nodes for each stage and (2) creating edges for every possible stall condition for each stage. Once a complete graph is constructed, reductions are typically possible to reduce complexity without sacrificing precision.

will discuss a procedure for developing such a model given a specification of the processor pipeline. We will use the relatively sophisticated processor Figure 3.3(a) of to illustrate the procedure. This processor is still simplified compared to real processors (especially in the memory system) but is complex enough for illustrating the procedure.

We start below with an overview of the three-step process for constructing a graph model.

- **Create nodes.** Create two nodes for every pipeline stage: one to signify the dynamic instruction entering the stage (*e.g.,* F_e) and one to signify exiting that stage (*e.g.,* F_x).

- **Create edges.** For each pipeline stage, identify all of the reasons that a dynamic instruction may be "stalled". In other words, identify the machine and program dependences that inhibit the dynamic instruction from either entering or exiting each pipeline stage. These dependences will be represented by edges between nodes in the graph.

- **Simplify.** As will be clear from the example, some nodes and edges created above will be redundant and can be removed.

- **Add latencies.** Label each edge with latencies. For each edge, the latency is the minimum number of cycles after the source node fires that the target node can fire.

As will become clear through an example, this procedure does not require an architect to think very hard to construct a graph model. In fact, all of the information that is required about the machine is only a subset of what is necessary to write a typical processor simulator.

Create nodes. For the example processor of Figure 3.3(a), we have eight pipeline stages and sixteen nodes (Fe_e, Fe_x, IB_e, IB_x, etc), as shown in Figure 3.3(b).

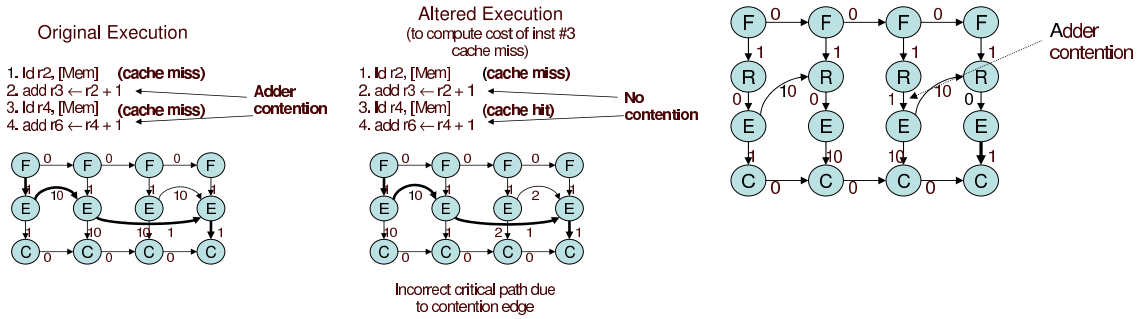
Pipe Stage	**Stall Causes**
Fe	Fetch Buffer Full ($IB_x \rightarrow Fe_e$), Limited Fetch Bandwidth ($Fe_x \rightarrow Fe_e$)
IB	Instruction Queue Full ($IQ_x \rightarrow IB_x$), Reorder Buffer Full ($Co_x \rightarrow IB_x$), Limited Fetch Buffer Output Bandwidth ($IB_x \rightarrow IB_x$)
Re	None
IQ	Data Dependencies ($IQ_x \rightarrow IQ_x$), Functional Unit Contention ($IQ_x \rightarrow IQ_x$), Memory Request Queue Full ($IQ_x \rightarrow IQ_x$)
Rg	None
Ex	None
Da	None
Co	Limited Commit Bandwidth ($Co_x \rightarrow Co_x$)

Table 3.1: **Machine Dependences and Corresponding Graph Edges for Each Pipeline Stage.**

Create edges. The stall culprits for each stage of the pipeline are shown in Table 3.1; these are directly translated into edges. For example, a dynamic instruction can be stalled in the IB stage due to the issue queue being full (resulting in a $IQ_e \rightarrow IB_e$ edge), the reorder buffer being full ($Co_e \rightarrow IB_e$), or fetch buffer output bandwidth being saturated ($IB_e \rightarrow IB_e$).

Simplify. As you can see from Figure 3.3(b), some nodes have only one incoming and one outgoing edge and, thus, are not useful to have in the graph (unless including them makes the graph more readable for humans.) They can be removed to yield more simpler graph.

Add latencies. The latencies are design parameters determined from a machine's implementation. $IQ_x \rightarrow IQ_x$ edges will be labeled with execution latency of operations. $Fe_x \rightarrow Fe_e$ edges will have a label of one cycle since instruction fetch is delayed by one cycle when fetch bandwidth is saturated.



(a) Code snippet illustrating modeling problem. **(b)** An alternative modeling.

Figure 3.4: **Certain types of resources cause problems with graph analysis.** While it is possible to construct a graph model for any microexecution that can be expressed as a waterfall diagram, certain resources do present problems for some forms of graph analysis. In **(a)**, we attempt to find the performance effect of a cache miss by observing the change in the critical path when we convert it's *graph representation* from a miss to a hit (by reducing the latency on the edge). Our analysis yields inaccurate results, however, since we did not account for the reduced adder contention after the miss is removed (the contention edge is still present in the graph.) In **(b)**, we use a different modeling of adder contention, where it is included as a latency that delays execution as opposed to a dependence edge. Empirically, we have found this representation typically reduces the error introduced by side effects.

3.3.1 Sources of modeling error

Side effects

There are certain types of resources which cause modeling difficulties when using the above procedure. They occur when a modification to a processor execution produces unanticipated "side-effects" as far as the graph modeling is concerned. To understand this effect better, consider the task of finding the performance cost of a cache miss using the graph, as discussed above. Our approach would be to measure two critical-path lengths: (1) with the cache miss included in the graph and (2) with the cache miss's latency reduced to that of a cache hit. The difference between these two lengths would by the cache miss's cost.

Notice that the only modification we made to the graph to obtain the critical-path length

of (2) above was to reduce the latency of one edge in the graph, corresponding to the cache miss. Problem sometimes arise with this approach when reducing a load's latency in the real execution has effects beyond simply this latency change. For example, reducing the latency might increase the rate at which instructions reach functional units, thus increasing the effect of functional-unit contention on execution time. If the graph is constructed such that this increased contention is not accounted for, some inaccuracy in the cost computation will result. An example of this effect is illustrated graphically with a code snippet in Figure 3.4(a).

It is important to point out at this point that most "side effects" will, in fact, be accounted for naturally in the graph model. For example, reducing the latency of the cache miss might also decrease the number of ROB stalls that occur. This effect is accurately modeled by the CD edges in the graph: fewer of them will be critical in the altered graph. Side-effects can cause inaccuracies in the analyses for one of the following two reasons:

- **Incomplete Modeling.** When designing the model, it is often convenient to leave out some details, both for efficiency and ease of reasoning about the execution. If a resource, such as functional-unit contention, is not explicitly modeled in the graph, its performance effect in the altered execution would obviously not be captured.

- **Dynamic Control Policies.** It is common for processors to have one or more control policies in the hardware that make choices about how to allocate resources to instructions. For example, an instruction scheduler might choose the oldest data-ready instructions to execute each cycle (such that the newer data-ready instructions have to wait.) In this case, altering the execution might change the order in which instructions become data-ready, which, due to the scheduling policy, might change how long they have to wait for functional units. Many

other examples exist, such as cache-replacement policies, memory bus schedulers, and cluster assignment policies. It is very difficult to incorporate such policies into graph analyses.

For side-effects caused by dynamic control policies, we have employed two approaches in our work. The first is to implement the policy as part of the graph analysis. If the policy would make different decisions due to the altered execution, we reflect those decisions by altering the appropriate edges and latencies in the graph. The upside of this approach is that there are no inaccuracies in the measurements. Unfortunately, this increase in accuracy comes at the expense of a less efficient analysis.

An alternative is to make some graph alterations to mitigate the effect of side-effects without degrading efficiency. For example, Figure 3.4(b) shows contention measured, not as a dependence edge between instructions, but as an intra-instruction constraint that delays the firing of the execute node. With this modeling, there would not, for example, be an extraneous dependency between instructions two and four in the altered execution of Figure 3.4(a), but instead instruction four's execution will simply be delayed by one cycle longer than it should be. In our experience, extraneous dependencies are typically much more damaging to precision than a few extra cycles of latency. In the next section, we discuss some approached to validating graph models, which can help quantify the errors due to side effects experienced with various modeling alternatives.

Bad-path instructions

Another potential source of inaccuracy in our modeling is the lack of accounting for instructions that were executed speculatively and then later squashed. The most common example of this is when a branch is mispredicted, causing instructions on the wrong path to be fetched and

executed.

Note that the model does account for the cost of branch mispredictions (with EF edges), it's only the resource contention caused by bad-path instructions that is ignored. For example, if a bad-path instruction used an available functional unit, making a good-path instruction (from before the branch) wait longer than it would otherwise, performance is affected, but our model does not capture that effect. Fortunately, we have found empirically that the resource contention imposed by bad-path instructions is not significant and, thus, is safe to ignore.

The effect might be significant, however, in other types of machine organizations, such as a processor supporting SMT. If this is the case, bad-path instructions would need to be included either directly with nodes and edges or indirectly with some estimation of the contention they would impose on important resources. We have not studied this type of modeling in detail.

3.4 Validation

There are several reasons why a graph model should be validated before use. First, there may be important machine dependencies that were neglected during its development. Second, it is sometimes practical to abstract away many of the details of the processor, in order to obtain a simpler, more intuitive graph model. Finally, it is important to quantify the inaccuracies imposed by side effects and bad-path instructions (see Section 3.3.1) on our analyses. By validating the model, we can reach a level of assurance that the measurements taken from it are reasonably accurate.

In our work, we have used two forms of validation. The first is a *small-signal analysis* that introduces small changes to the microexecution (such as adding a cycle of latency to an event) and observes whether the performance effect of the changes as observed by the simulator match those

taken from graph analysis. The second is a *large-signal analysis* that tests the model's capability to determine the effect of large changes to the microexecution (such as removing all cache misses). Both forms of validation are useful since, while the second is a more intense test of the model, the first is more representative of the types of graph analyses most useful for optimization engines.

3.4.1 Small-signal Validation

Our simplest validation uses the simulator to perform the following two measurements.

- Decrease the execution latency of every dynamic instruction (by one cycle) that is found by the graph **not** be on the critical path. Since only non-critical instructions are sped up, there should be no effect on execution time.

- Increase the execution latency of every dynamic instruction (by one cycle) that is found to be on the critical path. This should result in a large increase in execution time.

Note that these measurements are merely illustrative of how a small-signal validation experiment could be designed. Since the graph model includes a lot of edges, not just those that represent execution latencies, adjusting latencies that correspond to other edges could also be beneficial.

Also, as a practical concern, since some instructions have an execution latency of one cycle, and our simulator does not support execution latencies of zero cycles, we established a baseline by running a simulation where all the latencies were increased by one cycle. The critical path from this baseline simulation was written to disk. We then ran two simulations that each read the baseline critical path and decreased all critical (non-critical) latencies by one cycle, respectively.

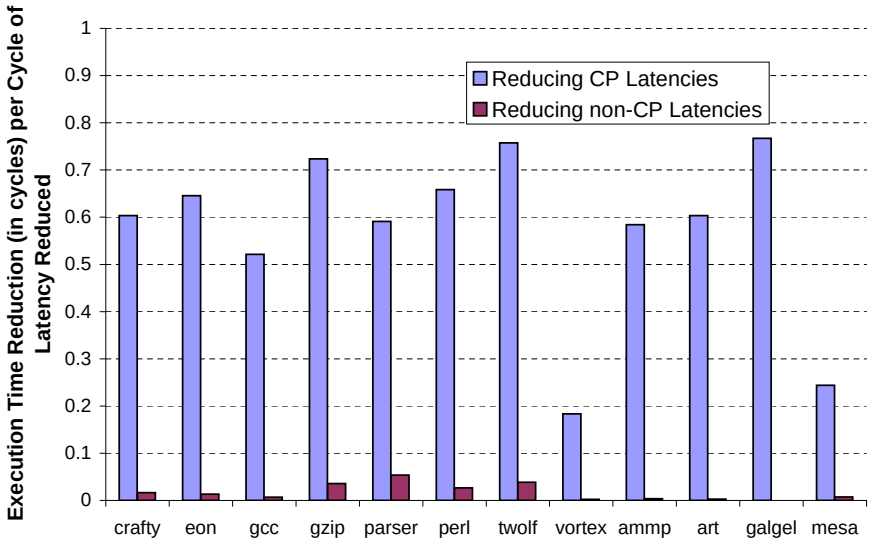


Figure 3.5: **Validation of the critical-path model** Comparison of the performance improvement from reducing critical latencies vs. non-critical latencies. The performance improvement from reducing critical latencies is much higher than from non-critical latencies, demonstrating the ability of our model to differentiate critical and non-critical instructions. The simulator methodology used for the experiments is described in Section 7.1.

The results of the experiment over the SPECint benchmarks for a simple three-node model is shown in Figure 3.5, where each speedup is shown as cycles of execution time reduction per cycle of latency decreased. The most important point in this figure is that the performance improvement from decreasing critical path latencies is much larger than from decreasing non-critical latencies. This indicates that our model, indeed, identifies instructions critical to performance.

Note that even though we are directly reducing critical path latencies, not every cycle of latency reduction turns into a reduction of a cycle of execution time. This is because reducing critical latencies can cause a *near-critical* path to emerge as the new critical path. Thus, the magnitude of performance improvement is an indication of the degree of dominance of the critical path. From the figure, we see that the dominance of the critical path varies across the different benchmarks. To get the most leverage from optimizations, it may be desirable to optimize this new critical path as well.

Finally, there is a very small performance improvement from decreasing non-critical latencies. This is the result of imprecise modeling, resulting from one of the three sources mentioned at the beginning of this section.

In our experience, developing an accurate graph model of a complex processor does require repeated validation and debugging. A practical technique to quickly identify problems with a model has been to compare the following two graphs, edge by edge: (1) the graph that has been altered to idealize a resource (or, for example, decrease the latency of non-critical instructions) and (2) a graph produced by the simulator that has been modified to idealize a resource. If the graph model is a perfect representation of the simulator, these two graphs should be identical. If there are discrepancies, it could be due to side effects (see Section 3.3.1) or important machine dependencies left out of the model. Tracing microarchitectural behavior up to one of these discrepancies will typically unveil one or more sources of inaccuracy in the model.

3.4.2 Large-signal Validation

The validation described above involved only one cycle adjustments to latencies. The overall change to the execution was still rather substantial, since many one cycle adjustments were made; and the characteristic of reducing many latencies by a small amount is a good match for the performance effect of many dynamic online optimizations. Nonetheless, it is a relatively modest change to the execution compared to some of our more advanced analyses. For a more rigorous test, we perform large perturbations to the model and see if those changes match what we would expect based on simulator output.

For example, one such validation is to increase every instruction's latency (in the simulator) by the amount of *slack* that the model determined that it had. If the slack computations were

correct, the execution time of the program should not increase at all.

Another of the large-signal validation experiments involve *idealizing* an entire set of processor resources both in the simulator and on the graph. For example, one measurement may be turning all cache misses into hits on the graph, measuring the critical-path length change this induces; and then making the same modification in the simulator, observing the execution time reduction. If the graph and simulator measurements are close in value for a large range of idealizations of this sort, we can have some assurance that the model is producing accurate *cost* results.

We will present experiments for each of these types of validation as the concepts of slack, cost, and interaction cost is presented later in the thesis.

3.5 Summary

In this section, we reasoned what the fundamental characteristics of the critical path should be in a microprocessor. This exercise yielded one characteristic in particular, namely performance sensitivity. From this foundation, many implications were drawn, culminating in a dependence graph of the program's microarchitectural execution. We discussed a procedure used to construct such a graph starting from a traditional pipetrace of a program's execution (which is often represented by a waterfall diagram). With this procedure, the critical path can be found for any program execution for which a pipetrace exists.

For analyses that are more sophisticated than simply finding the critical path, the dependence graph that is found using a pipetrace must be augmented with other dependence constraints that exist in the processor — even if they are not manifested in a particular execution. When building such a graph, a tradeoff exists between accuracy and complexity (in terms of the number of nodes

and edges). Besides complexity, accuracy can also be compromised by aspects of some microarchitectures that cannot easily (if at all) be captured in the types of graph models used in our work. To gauge the effectiveness of a particular graph model, we have devised validation procedures which provide simple empirical measurements of accuracy.

Chapter 4

Interpreting a Program's Critical Path

In the last section, we presented a model of program performance that enabled the determination of which events are critical and which are not. While this information is useful for some purposes on its own, more refined metrics are necessary for most practical applications. For example, compiler use of criticality is complicated since an instruction as a whole cannot be simply stated as critical or non-critical, since criticality is associated with much more basic micro-operations. A second consideration is that some events are "more critical" than others, while some non-critical ones are very close to being critical. Even more challenging, the performance effect of multiple events often *interact*. For example, when two cache misses are issued to the memory system simultaneously, both could be equally critical. (More often, one of the misses is critical and the other is "slightly" non-critical but, as far as optimizations are concerned, both misses should be targeted.) In this section, we will extend our work on criticality to provide a more complete picture of performance to an analyst.

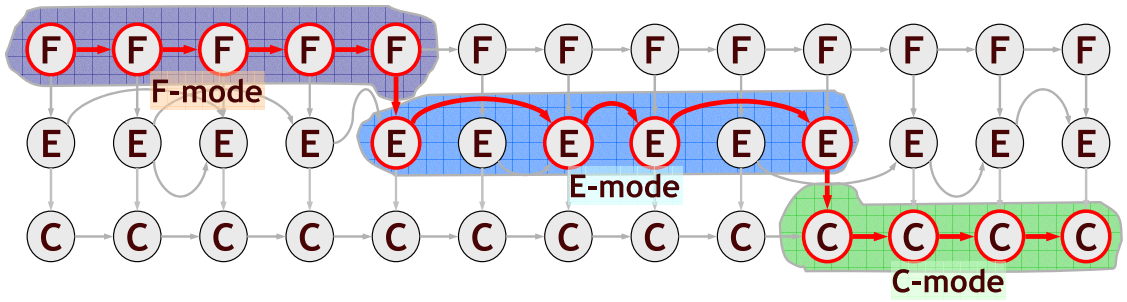


Figure 4.1: **Criticality modes.** The processor can be viewed as operating in a particular criticality mode in each cycle of execution, corresponding to which resources the critical path is traversing through in that cycle.

4.1 Criticality Modes

One useful way for an analyst to understand the criticality characteristics of a program execution is to think of various *modes* of operation, each corresponding to the various resources that could be limiting the processor at any given time. In this way of thinking, in each cycle of execution, the processor is in a particular mode and optimizations can be targeted for that mode. For example, when an execution is in "fetch mode", the instruction fetch resources are limiting performance. In this case, optimizations targeting the front end of the machine are most likely to be beneficial. Similarly, the execution could be in "execute mode", where resources such as ALUs and the instruction window are most important to performance.

An illustration of the criticality modes using the simple three-node graph model is shown in Figure 4.1. The critical path determines which mode the processor is in at any given time. If the critical path is traversing *F* nodes, the processor is in *F* mode, meaning the instruction fetch and dispatch resources are limiting performance. Similarly, if the critical path is traversing *E* nodes, the processor is in *E* mode and execution resources (*e.g.*, ALUs, issue logic) are limiting performance. Finally, if the processor is in *C* mode, the commit logic is most important to performance.

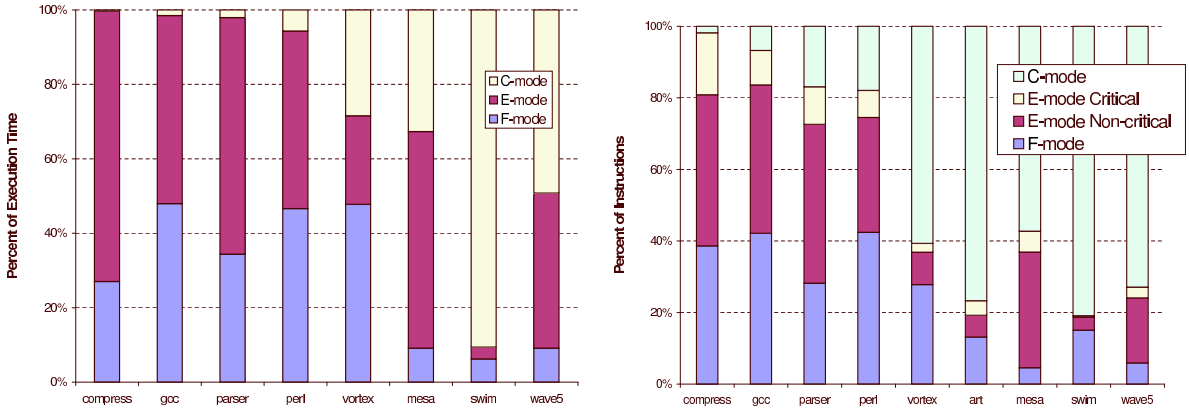
Another way to think of modes is that a particular dynamic instruction can be critical for different reasons (*i.e.,* due to one or more of the phases of its processing). For example, it can be fetch-critical, execute-critical, or commit-critical, as illustrated in the figure. Moreover, when a program execution is in *execute mode*, some of the instructions will likely not be critical at all (since the critical path traverses EE edges, which will likely "skip" over some instructions).

We show in Figure 4.2a the fraction of *execution time* spent in each of the three modes for a few familiar benchmarks. In this figure, the number of *cycles* the program spends in each mode is shown. Figure 4.2b shows the fraction of *dynamic instructions* that are in each mode. The latter figure gives an indication of the number of opportunities for optimizing each mode are available.

We can use this type of information to design optimizations where the most *leverage* exists. For instance, in the integer applications, where *F* mode is significant, performance could be improved through a more powerful fetch unit—maybe a trace cache would be of benefit. In many floating point benchmarks, *C* mode dominates. In these, a larger re-order buffer—to permit the exploitation of more parallelism—could be of significant benefit. Value prediction has the potential to break *EE* edges. Hence, the technique is most applicable for benchmarks that spend a lot of execution time in *E* mode.

We show in Figure 4.2b the fraction of *instructions* in each of the three modes. For this measurement, the *E* mode is broken into two components: the instructions that are on the critical path and those that are not—they are "skipped" over by *EE* edges.

This breakdown illustrates the power the critical path has to enhance processor optimizations. Instruction issue scheduling, for instance, is only important for instructions in *E* mode, dispatch-critical and commit-critical instructions will not benefit from a priority-based issue pol-



(a) Execution Time Breakdown (b) Instruction Count Breakdown

Figure 4.2: **The critical-path model can be used to compute precise performance breakdowns.** (a) shows the contribution of each mode to execution time for each benchmark, suggesting what optimizations would be best applied. (b) shows the percentage of instructions that are limited by each mode of operation. Optimizations should focus on those instructions that will impact performance. The simulator configuration for these experiments is described in Section 7.1.

icy. Furthermore, those instructions that are critical in *E* mode should be given the highest priority.

For value prediction, it only makes sense to make predictions to break critical *EE* edges. Thus, value prediction should only be applied to execute-critical instructions. Making predictions to other instructions would only needlessly risk a misspeculation. After a value prediction is made that breaks the critical path, however, a near-critical instruction may emerge on the new critical path. This new instruction could then benefit from value prediction.

The notion of near-criticality is important for a large number of analysis applications. For example, in performing dynamic reconfiguration and energy optimizations it is often important to know the minimum number of resources needed for processing without increasing the length of the critical path. In the next section, we discuss metrics for quantifying how critical or non-critical an event is, to provide a general analysis methodology for a broad range of applications.

4.2 Criticality of a Single Event

Inherent in the notion of near-criticality is that the degree of criticality of one event typically depends on the latency and criticality of other events. These interdependencies complicate performance understanding considerably, since, in principle, every event in the program execution could influence the criticality of every other event. To start the discussion, we'll ignore the exponential number of interdependencies and focus on the much simpler problem of quantifying the criticality of a single event. We'll explore the more general notion of criticality in Section 4.3.

As we know by now, the latency of an event is not sufficient to indicate to what degree an event is critical. Latency is directly related to criticality, however. Consider the dependence graph of Figure 4.3a. The circled edge weight represents the latency of a data-cache miss, at 9 cycles. As can be seen, this cache miss's latency is not on the critical path. In other words, the load has some amount of *slack*, such that the latency could be increased without increasing the execution time of the entire program.

Now, consider increasing the latency of the cache miss from 9 to 11 cycles, as shown in Figure 4.3b. This change of latency puts the cache miss on the critical path; in other words, the longest path in the graph now includes the cache miss. This means the cache miss has a *cost* to performance, in that decreasing its latency would decrease the execution time of the entire program.

The relationship between criticality and latency can be illustrated graphically by a chart, Figure 4.3c. In the example of the cache miss, a latency of ten cycles is the minimum that places it on the critical path. At this latency, this path is tied in length to another path flowing through the branch misprediction, completely hiding the latency of the cache miss — or, in other words, the miss has a cost of zero. Each cycle of additional latency increases the cache miss's cost by one

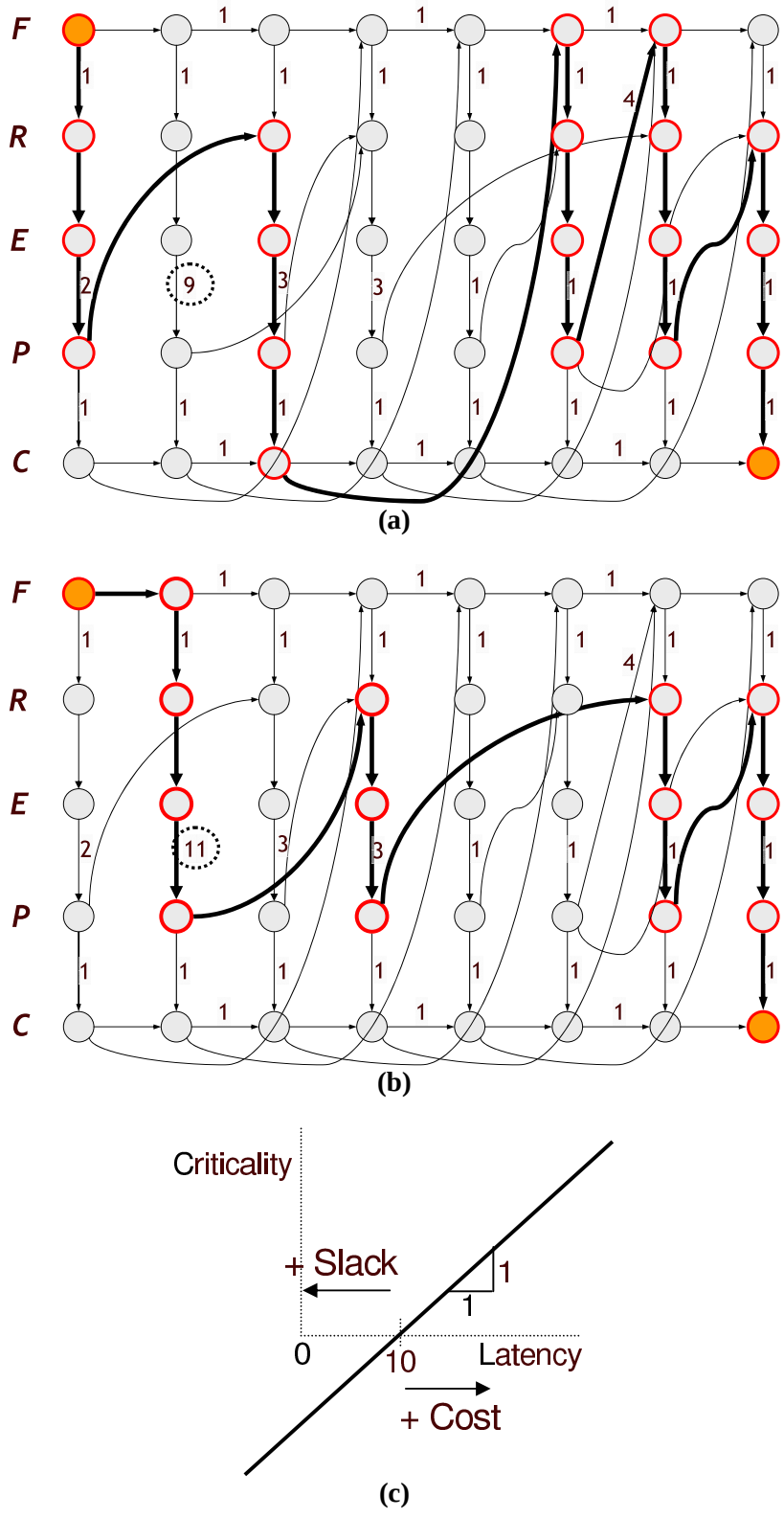
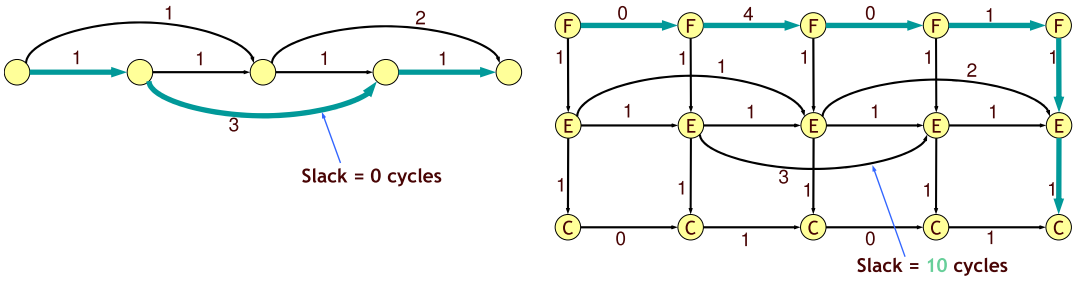


Figure 4.3: **Relationship between slack and cost.** In **(a)** the circled cache miss latency is off the critical path. By increasing the latency by a couple of cycles (see **(b)**), the critical path changes, placing the latency on the critical path. In **(a)**, the cache miss has one cycle of slack. In **(b)**, it has a cost of one cycle. The relationship between the latency, slack, and cost of the cache miss is shown in **(c)**.



(a) Slack without resource modeling. **(b)** Slack with resource modeling.

Figure 4.4: **Slack with and without resource modeling.** Modeling processor resources exposes more of the slack inherent in the execution than can be observed by a data dependences alone.

cycle; each cycle reduced (below 10 cycles) increases the cache miss's slack.

Definition of slack. The notion of slack is particularly useful to designers and compiler optimizers. The reason is simple: resources allocated to events that are far from critical can be re-allocated to critical events. Alternatively, slowing down the processing of those events can often lead to energy savings.

At this point we would like to note again the importance of modeling resources when determining the amount of slack that an event has. For an example, consider the *data-dependence only* graph of Figure 4.4a. Here, the indicated execution event has no slack, since it is on the critical path of the execution. If resources are also modeled, however, that same event has several cycles of slack (see Figure 4.4b). In general, modeling resources will increase the slack observed on execution events since the critical path often traverses resource edges.

In the computer architecture community, there is not a consistently used formal definition of slack. For our work, we charactered two different notions, each of which can be useful in different contexts. They are *local* and *global*:

The **local slack** of a dynamic event e is the maximum number of cycles the processing

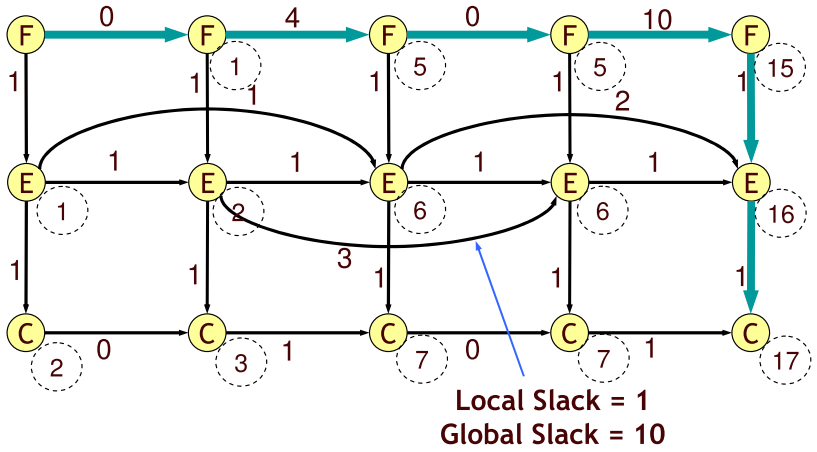


Figure 4.5: **Global vs. Local Slack.**

of e can be delayed *without delaying **any** subsequent events*. In other words, delaying an event by its local slack does not change which edges are last-arriving. For an example, consider the graph of Figure 4.5. In the figure, each node is annotated with it's "fire" time, *i.e.*, the cycle at which all dependences into that node have been resolved. The fire times make clear that the labeled execution edge has a local slack of only one cycle: increasing it's latency by more than one cycle would delay the fire time of the target node of the dependence.

From our measurements, approximately 20% of instructions have local slack greater than five cycles (see Section 7.2.1). Local slack is conservative because it prevents delaying any event in the program. To avoid impairing the overall execution, however, it suffices to ensure that the program completes in the original number of cycles. This more aggressive notion is captured by global slack.

Global slack of a dynamic event e is the maximum number of cycles the execution of e can be delayed *without delaying **the last** instruction in the program*. In other words, delaying an event by its global slack does not change what edges are on the critical path. From Figure 4.5, we can see that global slack offers much more optimization potential than local slack. In fact, from

our measurements, approximately 75% of instructions have global slack greater than five cycles (as opposed to 20% for local slack). Global slack serves as an upper bound on the amount of tolerable delay, since it is the maximum amount a particular instruction can be delayed without increasing execution time.

The difficulty with global slack is that, in contrast to local slack, a single cycle of slack may be shared among many events. In other words, there are interactions between the global slack of various events. We can delay every event in the program by its local slack without increasing execution time. The same is not true for global slack. We will address this issue in detail in Section 4.3.

Definition of cost. From a high level, the cost of an event e could be defined as the number of execution cycles that should be blamed on e. The issue of how to assign blame for execution cycles presents some subtle issues, however, which affect the definition of cost.

For example, consider the criticality of the edge representing the branch misprediction in Figure 4.6a. How should the cost of the misprediction be defined? One possibility is to define it as the execution time reduction that occurs when the edge's latency is reduced to zero. Notice, though, that even if the edge's latency is decreased to zero, the critical path still flows through the branch misprediction. This means that if the latency could somehow be made negative, execution time would decrease even more.

The relationship between the branch mispredict edge's latency and criticality is shown graphically in Figure 4.6b. The branch becomes non-critical if its latency reduces to less than -5 cycles. In a theoretical sense, then, we would say the cost of the branch in Figure 4.6a is its latency, 4, minus -5 for a total of 9 cycles, since the maximum possible reduction of execution time from

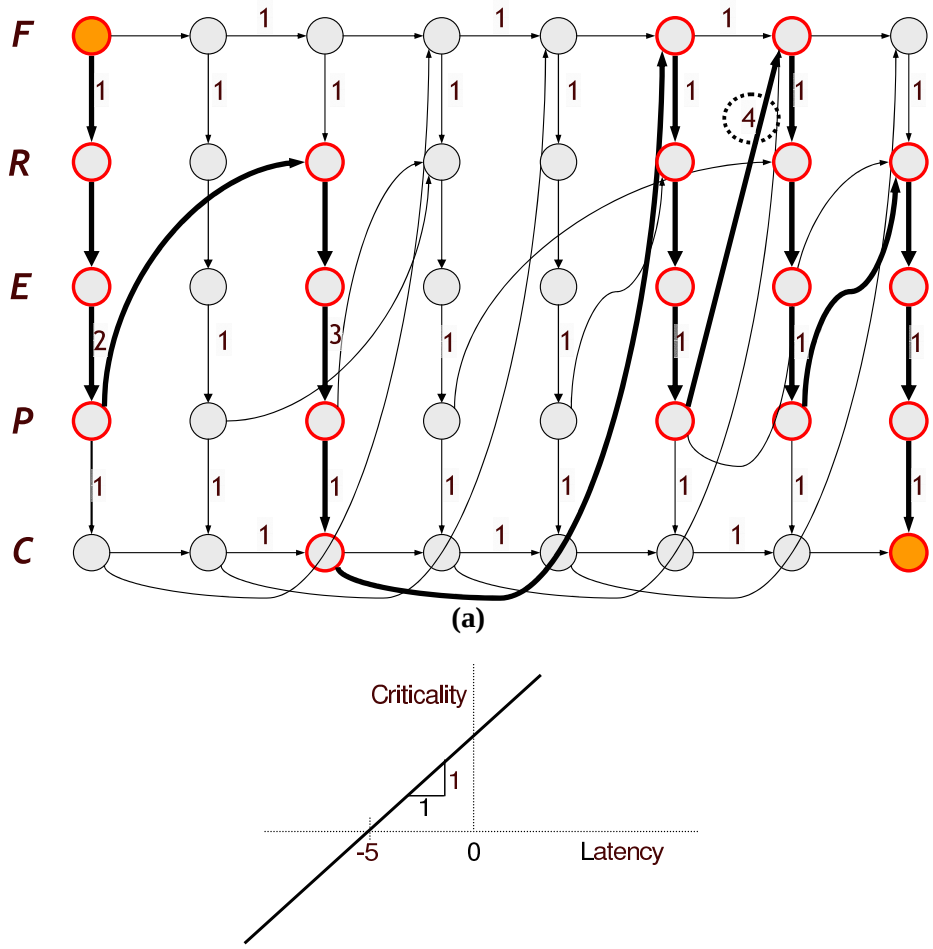


Figure 4.6: **Relationship between slack and cost for a branch mispredict.** The latency of a branch misprediction latency is circled in **(a)**. The relationship between it's latency and it's criticality is plotted in **(b)**. Notice that the branch misprediction remains critical (with criticality above the x-axis) until it's latency is below negative five cycles.

event type	**how to idealize in a simulator**
Icache, Dcache, TLB misses	turn misses into hits
ALU operation	give ALU zero cycle latency
Fetch,Issue, or Commit Bandwidth	use infinite bandwidth
Branch mispredict	turn mispredicts into correct predictions
Instruction window	use infinite-sized window

Table 4.1: **Idealizing events.** Listed are techniques to idealize a few of the events studied in this paper.

optimizing the branch is 9 cycles.

To generalize, the ultimate cost of any event e is the execution time reduction achieved if it's latency is reduced the maximum amount, or, in other words, to negative infinity. Of course, it is not possible in real life to make a latency negative. It is often possible, however, for an optimization to change the structure of the graph to achieve the same effect. In the case of the branch misprediction edge, an optimization that makes the prediction correct would remove the dependence entirely, which has exactly the same effect as reducing the latency to negative infinity.

To form a definition that is applicable across all types of events, we define the cost of an event as the execution time reduction obtained when the event is *idealized*, where "idealized" means that the performance impact of the event was reduced to the greatest extent possible for the analysis application (examples of this will be presented shortly). Formally, let e be an event, t be base execution time (nothing idealized), and $t(e)$ be execution time with e idealized. Then, the cost of e, $cost(e)$, is defined as

$$cost(e) \stackrel{\text{def}}{=} t - t(e)$$

The cost of an event can be naturally generalized to an *aggregate* cost of a set of dynamic events S. This allows us to compute, for example, the cost of a cache as the total speedup when all cache misses are idealized.

It is often the case in practice that it is more useful to define an "idealization" to be something less dramatic than setting an edge's latency to negative infinity. For example, a load prefetching optimization might be able to turn a cache miss into a cache hit, which would result in a reduction of latency on the graph from a cache miss latency (*e.g.,* 12 cycles) to a cache hit latency (*e.g.,* 2 cycles). For this application, it would make sense to define the cost of a cache miss to be the performance improvement when the latency is reduced by ten cycles. A number of example idealizations for practical applications is given in Table 4.1.

4.3 Degree of criticality of multiple events

As mentioned above, there is much more to performance analysis than just the slack or criticality of a single event. When events must be considered simultaneously to understand performance, we say those events *interact*. The simplest example of an interaction is two cache misses starting simultaneously and being serviced in parallel. The cost of either miss, as defined above, is zero: removing one miss or the other does nothing to performance. To understand the performance effect of these misses requires an analysis of interactions.

Interactions can be much more complex than in the case of two parallel cache misses. For one, the two misses may only partially overlap, giving the two misses each a small positive cost; or, if a third event occurs in parallel to both misses, one or both could have a positive slack. Secondly, the interacting events may be non-homogeneous, such as when an icache miss occurs at the same

time as an ROB stall. Finally, events do not have to occur simultaneously in order to experience interactions. The simplest example is the one given in Section 4.2 when distinguishing between local and global slack: if one event e_1 is dependent on another e_2, e_1 and e_2 may both "share" the same cycles of slack.

We have developed two ways of accounting for interactions in a parallel system:

- **Quantifying Interactions.** To maintain maximum flexibility in how interactions are to be exploited, we must explicitly quantify them. The quantification consists of measuring the effect that several events together have on execution time and comparing that to the execution-time effect that the individual constituent events have. In this way, the interaction effects can be explicitly measured. The disadvantage of this approach is that there are an exponential number of interactions in a typical program run, necessitating heuristics that reduce the amount of information that must be observed when making optimization decisions. Nonetheless, well-chosen heuristics that focus on a small subset of events that are important for a particular analysis application can make explicit quantification practical.

- **Apportioning.** An alternative method to account for interactions is to isolate the analyst or optimizer from their effect. We can do this by pre-apportioning the global slack or cost to individual events, essentially converting them to local metrics which can be considered in isolation. In other words, optimization decisions would be made without considering interactions explicitly, relying, instead, on an pre-apportioning policy to assign each event a reasonable share of the slack or cost.

4.3.1 Quantifying Interactions

To start off, consider the above example of the two cache misses. While the cost of the individual cache misses are zero, the *aggregate* cost of both cache misses, obtained by measuring the execution time reduction from idealizing both c_1 and c_2 simultaneously, would be large. By knowing this aggregate cost, denoted $cost(\{c_1, c_2\})$, the program optimizer would know that while prefetching only one load would give little benefit, prefetching both would give significant benefit. We term this phenomenon, where $cost(\{c_1, c_2\}) > cost(c_1) + cost(c_2)$, a *parallel interaction*.

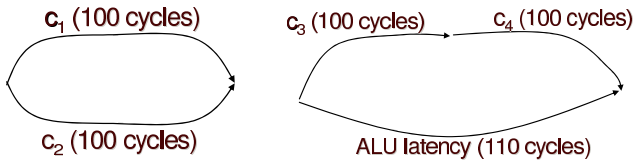
Perhaps less intuitively, it is also possible for the opposite parallelism-induced effect to occur, where $cost(\{c_1, c_2\}) < cost(c_1) + cost(c_2)$. One example is if two *dependent* cache misses, each with 100 cycle latency, both occurred in parallel with 100 cycles of ALU operations. In this situation, prefetching both provides no more benefit than prefetching either one alone, implying that a program optimizer would save overhead by performing only one prefetch. In general, this type of interaction can occur between two events A and B if they are in series with each other, but in parallel with some other event (or events) C. We call this phenomenon a *serial interaction*, since the two interacting events occur in series.

In summary, for two events e_1 and e_2:

$$cost(\{e_1, e_2\}) = cost(e_1) + cost(e_2) \Leftrightarrow \textbf{\textit{Independent}}$$

$$cost(\{e_1, e_2\}) > cost(e_1) + cost(e_2) \Leftrightarrow \textbf{\textit{Parallel Interaction}}$$

$$cost(\{e_1, e_2\}) < cost(e_1) + cost(e_2) \Leftrightarrow \textbf{\textit{Serial Interaction}}$$



(a) Parallel interaction. **(b)** Serial interaction.

Figure 4.7: **Two types of interactions.** Dependence graphs can conveniently illustrate the two distinct types of interactions that can exist between events: *parallel* and *serial*. In **(a)**, the two cache misses c_1 and c_2 have a parallel interaction. Both cache misses would need to be eliminated to improve performance. In **(b)**, c_3 and c_4 experience a serial interaction. Here, eliminating either cache miss will reduce execution time by 90 cycles, but eliminating both will not improve performance any further.

It is often convenient to visualize interactions using dependence graphs, like the ones in Figure 4.7.

As we show in Chapter 7, interactions are common phenomena (after all, there is potential for interaction any time two events occur simultaneously). To inform the optimizer (automatic or human) of the "degree" of interaction, we define interaction cost. Let e_1 and e_2 be two events and $cost(\{e_1, e_2\})$ be the aggregate cost of both events. Then, the **interaction cost** of e_1 and e_2, denoted $icost(\{e_1, e_2\})$, is defined as the difference between the aggregate cost of the two events and the sum of their individual costs:

$$icost(\{e_1, e_2\}) \stackrel{\text{def}}{=} cost(\{e_1, e_2\}) - cost(e_1) - cost(e_2)$$

Thus, for a parallel interaction, $icost(\{e_1, e_2\})$ is the number of extra cycles an optimization that targets both events, instead of just one, could ever hope to benefit. In contrast, for a serial interaction, $icost(\{e_1, e_2\})$ would be negative, reducing the expectation for performance improvement from targeting both events.

The interaction cost of two sets of events, S_1 and S_2, is defined similarly, by replacing e_1 and e_2 with S_1 and S_2 in the above equation. Moreover, the interaction cost of more than two events (or sets) can be defined recursively. Formally, let $\mathcal{P}(U) \setminus U$ denote the proper power set of a set of events U (*i.e.*, all subsets of U except for U itself). Then the interaction cost of U is defined as the cost of U minus the interaction cost of each proper subset of U:

$$icost(\{\}) \stackrel{\text{def}}{=} 0$$

$$icost(U) \stackrel{\text{def}}{=} cost(U) - \sum_{V \in \mathcal{P}(U) \setminus U} icost(V)$$

Notice the power of $icost$: it characterizes the interaction between events in a single number, with straightforward interpretation. The sign indicates the type of interaction (positive for parallel, negative for serial); and the magnitude indicates the *degree* of interaction. An $icost$ of zero means the two events are independent and can be optimized separately.

Finally, if U above is the set of *all* events in an execution it follows that total execution time always equals the sum of the $icosts$ for the powerset of U. This implies that completely accounting for execution time requires all interaction costs to be considered.

Icost optimization strategies

There are several ways that interaction costs may be used by an analyst during design and optimization. The one we have used the most in our is in the construction of performance breakdowns.

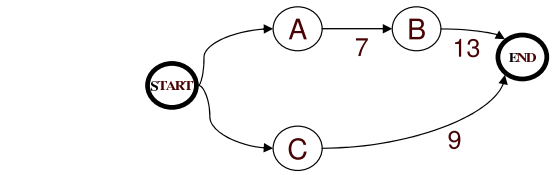
Performance breakdowns. A performance breakdown is a mapping of execution cycles to the events and hardware resources that are responsible (and should be blamed) for those cycles. They are often presented as stacked-bar charts in research papers, where each category (*e.g.*, cache misses, branch mispredicts) is sized to reflect it's overall contribution to execution time.

It is interesting to note that while it is very common for performance breakdowns to be used in research and design, the breakdowns that are used in practice are generally not a realistic depiction of performance in a processor that exploits parallelism. The reason is that, in general, several categories of events and resources could be responsible for a single cycle, but traditional performance breakdowns map each cycle to only one category. We seek to overcome this limitation using interaction costs.

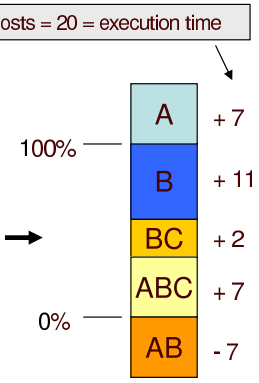
Figure 4.8 illustrates a breakdown that explicitly includes interaction costs. Notice that a complete breakdown of performance requires measuring all possible interactions between the breakdown categories. In practice, the less significant interactions could be grouped together to reduce the number of measurements necessary.

The table of interaction costs shown in Figure 4.8a provides a complete breakdown of execution time. Sometimes it is useful to visualize the results in something other than a table, however. Towards that end, we present one possible visualization in Figure 4.8b.

The most notable feature of this visualization is how we deal with categories that have negative values (due to serial interactions). In this case, we allow the stacked-bar chart to grow above 100% and below 0%. The chart grows above 100% because the total number of cycles allocated to the positive categories will exceed the execution time of the program. These extra cycles are offset by the negative categories, which we plot below the 0% axis. With this visualization, all interactions



Cat. Name	Expression for computing icost	Interaction Cost (icost)	Comments
A	cost(A)	7	All A's latency is critical
B	cost(B)	11	B is partially "hidden"
C	cost(C)	0	C is totally hidden
AB	cost({A,B}) - icost(A) - icost(B)	11 - 7 - 11 = -7	Optimize A or B, not both
AC	cost({A,C}) - icost(A) - icost(C)	7 - 7 - 0 = 0	A and C are independent
BC	cost({B,C}) - icost(B) - icost(C)	13 - 11 - 0 = 2	Bonus for optimizing A and B
ABC	cost({A,B,C}) - icost({A,B}) - icost({A,C}) - icost({B,C}) - icost(A) - icost(B) - icost(C)	20 - (-7) - 0 - 2 - 7 - 11 - 0 = 7	Bonus for optimizing all of A, B, and C



(a) Breakdown example

(b) One possible visualization

Figure 4.8: **Correctly reporting breakdowns.** The traditional method for reporting breakdowns does not accurately account for *all* execution cycles, since it attempts to assign blame for each cycle to a *single* event when sometimes multiple events are simultaneously responsible. We propose a new method that uses *interaction costs*, discussed in Section 4.3.1. In our method, each category corresponds to an interaction cost of a set of "base" categories.

are made apparent to the user (their sign is given by the location in the chart). For instance, the large negative contribution of AB in Figure 4.8 tells us optimizing A will help alleviate the costs due to B, since they tend to occur serially.

Interpreting Breakdowns. The cost of a single event is easy to understand: it is the maximum improvement possible by optimizing that event. If an optimizer could only choose a single event to optimize, it would pick the one with the largest cost (after taking the difficulty of implementing the optimization into account).

On the other hand, if an optimizer had the ability to improve several events, interaction costs become important. A parallel interaction between two events (*e.g.*, B and C in the figure) represents a "bonus" reduction in cycles for optimizing both together — this bonus cannot be derived from the individual event costs.

A serial interaction (*e.g.*, between A and B in the figure) can be applied in at least two ways by the optimizer.

- First, it tempers the expectation of performance improvement from optimizing both events: the total effect will be less than the sum of their individual costs.
- Second, it gives the optimizer a choice in what to optimize: the same set of cycles (equal in magnitude to the *icost*) will be eliminated if either A or B is optimized — in the example, seven cycles can be eliminated by optimizing either A or B, whereas four cycles can only be eliminated by optimizing B.

We make extensive use of the second application of serial interactions above in the case study of Section 7.3.1.

Optimization Heuristic. Although not explored in detail in this thesis, one way $icosts$ are useful is in choosing a set of events that should receive the most effort during optimization. Unfortunately, given an execution, *optimally* choosing a set of events to improve is an NP-complete problem. Nonetheless, interaction costs enable heuristics that may work well in practice, such as the one below, which assumes that the optimizer can choose N events to improve:

One of many possible optimization heuristics using interaction costs

1. Initialize the $expected\_benefit$ for each event to its individual cost.

   For each event e, $expected\_benefit(e) \leftarrow cost(e)$.

2. Pick event e_i with largest expected benefit to optimize.

   Of all events e, pick e_i such that $expected\_benefit(e_i)$ is largest.

3. For all events e_j that have a parallel interaction with e_i, increase their expected benefit by the magnitude of the $icost$.

   For all e_j, if $icost(e_i, e_j) > 0$, then $expected\_benefit(e_j) \leftarrow expected\_benefit(e_j) + icost(e_i, e_j)$.

4. For all events e_k that have a serial interaction with e_i, decrease their expected benefit by the magnitude of the $icost$.

   For all e_k, if $icost(e_i, e_k) < 0$, then $expected\_benefit(e_j) \leftarrow expected\_benefit(e_j) - abs(icost(e_i, e_j))$.

5. Until N events have been optimized, go to step 2.

This heuristic makes use of both parallel and serial interactions. Step 3 accounts for the "bonus" from parallel interactions by increasing the expected benefit of events that have a positive $icost$ with the event that has already been chosen for optimization. Step 4 accounts for serial interactions by tempering the expectation of improvement for events with a negative $icost$.

Slack Interactions. The same principles used for quantifying cost interactions above could be used for slack. A couple characteristics of slack appear to make another approach more attractive in practice, however. For one, slack only exhibits serial interactions; there is no graph that causes edges to have a parallel interaction in regards to slack. Second, we have found empirically that slack interactions typically involve many events, such that simple pairwise measurement would not be sufficient. Attempting to explicitly consider interactions involving tens of events while performing an optimization may be unwieldy.

These two observations make an alternative approach more appealing for most practical applications. We will discuss this "apportioning" style of interaction analysis in the next section.

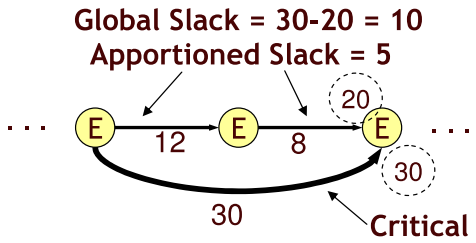


Figure 4.9: **Apportioning Slack.** The circled numbers are arrival times of the edges at the far left E node. The global slack of the shorter path is the difference in these arrival times. This global slack can be apportioned in many ways to individual edges along the shorter path. Here, each edge is apportioned five cycles.

4.3.2 Apportioning

Apportioning involves making pre-optimization decisions about how to allocate slack (or cost) to particular edges such that the user (or optimizer) is isolated from the effects of interactions. A simple example of apportioning in the context of slack is shown in Figure 4.9. Here, the global slack of each of the two top edges is 10 cycles, since that is the difference in arrival times of the critical and non-critical paths at the convergence point. It is not possible, of course, to delay each edge by 10 cycles, since that would delay the non-critical path to such a degree that it would become critical. If we *apportion* the global slack, however, giving 5 cycles to each edge, each edge can be delayed by it's apportioned slack without increasing execution time.

Of course, many valid apportioning policies exist. For the figure, giving 7 cycles to one edge and 3 to the other is also reasonable. The central goal is to maximize optimization opportunities such that, if each edge was delayed by its apportioned slack, all edges would be critical. In general, many apportioning policies will meet this goal. Which policy is most appropriate depends upon the optimization that will be performed.

More formally, let S be an assignment of some amount of slack (possibly zero) to each instruction in such a way that ***the last*** *instruction is not delayed*. Given an assignment of slack

S, the *apportioned slack* of instruction i is $S(i)$, *i.e.*, the slack assigned to i. The assignment can be arbitrary (as long as it does not delay the last instruction) and is intentionally left up to the apportioning policy.

Apportioning Policies. The choice of apportioning policy is highly dependent on the optimization that is to be pursued. Here are a couple policies we have experimented with in the past.

Five-cycle apportioning. One way to apportion slack is to attempt to give each instruction, say, five cycles of slack. This strategy might be useful if we wanted to know how many instructions could tolerate a long (non-uniform) bypass. From our measurements, approximately 75% of instructions have apportioned slack of five cycles. In other words, the execution contains a particular set of 75% of instructions that can be simultaneously delayed by five cycles. This surprising observation suggests tremendous optimization opportunities.

Latency-plus-one-cycle apportioning. Another apportioning strategy that we consider reflects a control policy for a constraint-aware processor that has a (power-efficient) ALU that runs at half the frequency of the other ALU. The goal of the control policy would be to maximize the number of instructions steered to the slow ALU, while maintaining the performance of a two-fast-ALUs machine. The corresponding apportioning strategy would be to maximize the number of instructions whose apportioned slack equals their original execution latency plus one cycle (so that they can tolerate the doubled latency of the slow unit plus some bypass overhead).

4.4 Summary

In this chapter, we have expanded upon our notion of criticality to distinguish not only whether a microprocessor event is critical or not, but also how critical or how far from critical the

event is. Furthermore, we have discussed two ways (quantifying and apportioning) for interpreting interactions between events that occur near each other in a parallel system. From these primitives, we have shown how accurate and complete performance breakdowns can be created.

Now that we have the underlying framework for our performance analysis, the rest of the thesis will deal with the more practical concerns of how to compute the metrics efficiently (Chapter 5), measure them in hardware (Chapter 6), and use them in productive ways (Chapter 7).

Chapter 5

Software Algorithms

In this chapter we will discuss in detail the algorithms used to efficiently compute the metrics discussed in the previous section. These algorithms are useful after the graph is constructed and available. Thus, for practical purposes, they are most useful in a simulator or after the graph is gleaned from performance counters (which would need to be enhanced via Shotgun Profiling, discussed in Section 6.3).

We will start by discussing a memory and computational efficient mechanism for computing the critical path of a program's execution. We will then present multiple ways to compute the more sophisticated metrics of slack and cost, depending on the particular demands of the analysis.

5.1 Computing criticality

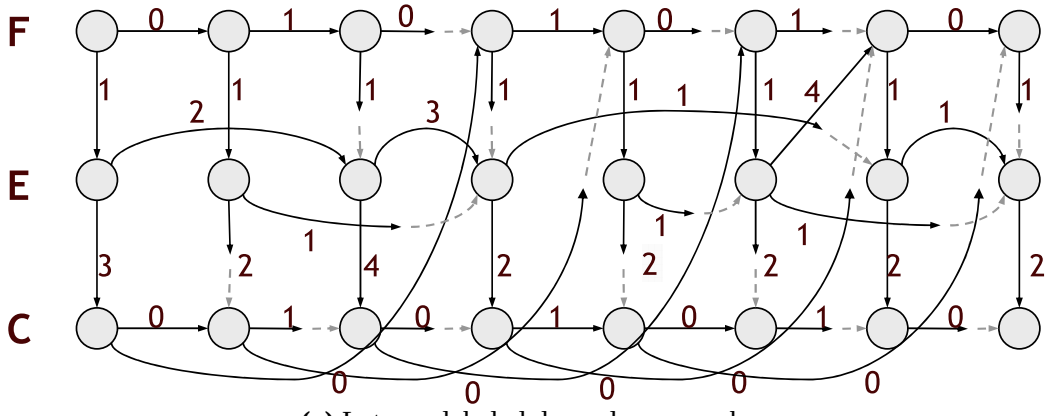
The critical path through the dependence graph is simply the longest path through the graph. Thus, the brute-force approach to calculating criticality is to enumerate all of the possible paths and pick the longest. Of course, a much more efficient algorithm is possible using a simple

topological sort. Our efficiency requirements are even more demanding, however. The problem is that the natural algorithm for finding criticality would require two passes, one for performing the topological sort and another for identifying the critical edges and nodes. This two-pass approach is rather expensive, however, since it requires buffering the entire graph, which would be very large for a long program execution run. In this section, we will describe our single-pass algorithm as it would be used in a simulator. In the next section, we will use some of the same principles to develop an efficient hardware implementation.

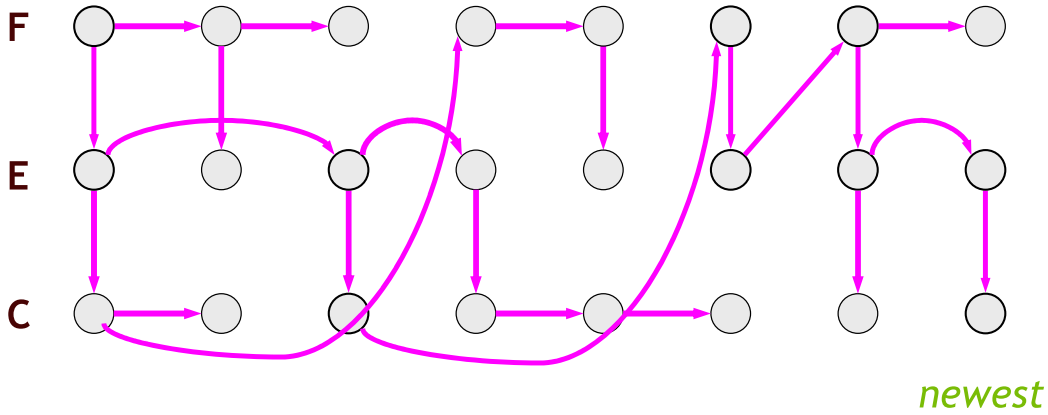
The first key insight is that an edge e that is on the critical path *must* be the **last-arriving** edge into its target node n. In other words, the critical edge must be on the longest latency path from the beginning of the program to its target node. This should not be a surprising conclusion since, by definition, a critical edge is also on the longest latency path through the entire program. If there was some longer latency path to n that did not include e, e could not be on the critical path.

So, we can find the critical path by building only a subset of edges of the graph: those that are last arriving. The critical path, then, is the chain of last-arriving edges from the first fetch node to the last commit node of the program. This chain can be found easily via a backward traversal from the end of the program. Figure 5.1 illustrates the procedure.

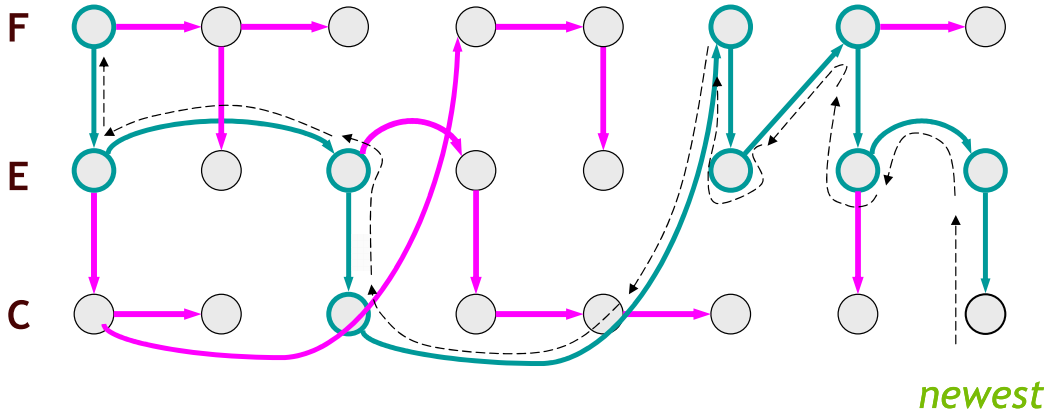
As described, the algorithm still requires buffering the entire graph. The second key insight that removes this requirement is that there are some edges in the graph that can be determined to be critical using local analysis, rather than the global last-arriving edge traversal described above. These edges, called *articulation edges*, effectively divide the large, whole-program-execution graph into many much smaller chunks, each of which can be operated on independently. Thus, only the portion of the graph up to the next articulation edge must be buffered in order to compute criticality.



(a) Latency labeled dependence graph.



(b) Only last-arriving edges are relevant.



(c) Backward traversal finds critical path.

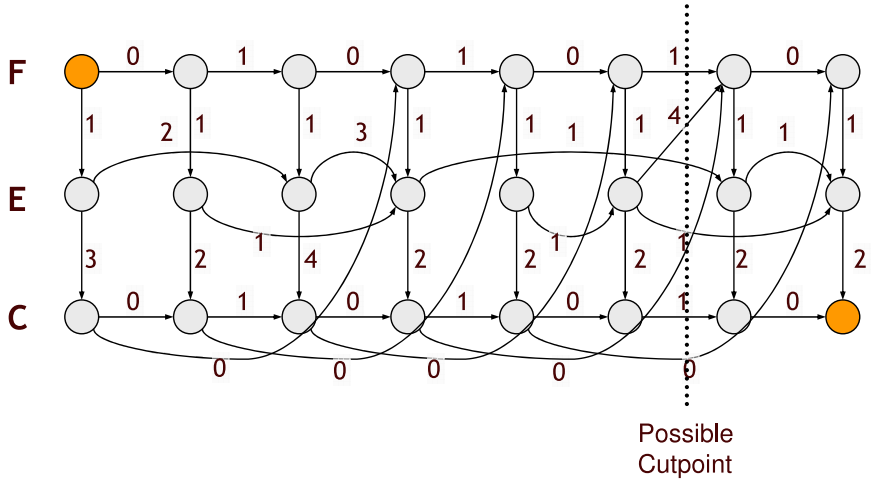
Figure 5.1: **Finding the critical path using last-arriving edges.**

If these articulation edges are frequent enough, the amount of buffering that is required would be modest.

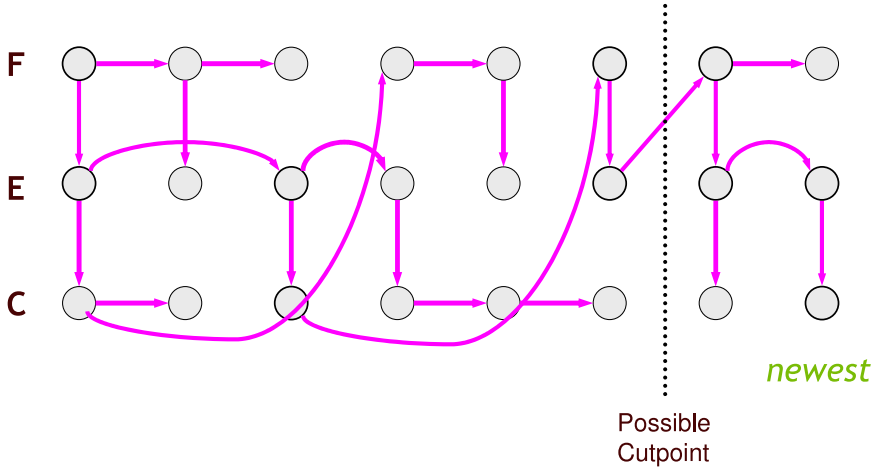
We employ two different strategies for identifying articulation edges. The simplest involves identifying *cuts* that divide all the nodes in the graph into one of two subgraphs. The cuts are made across a set of edges, as shown in Figure 5.2a; these edges are referred to as the *cut-set*. Since the critical path is a continuous sequence of last-arriving edges from the beginning of the program to the end, one of the edges in this cut must be critical. Furthermore, we know that every critical edge is also a last-arriving edge. Thus, if there is only one last-arriving edge in the cut-set, that edge must be critical. Since identifying whether an edge is last-arriving can be done without buffering the entire graph, this edge is an articulation edge for our analysis.

In practice, we have found branch misprediction (EF) edges are likely candidates for articulation edges. (In other words, a cut across an EF edge often yields a cut-set with only the EF edge as last-arriving.) Using this technique, it is typically possible to identify an articulation edge every few hundred instructions.

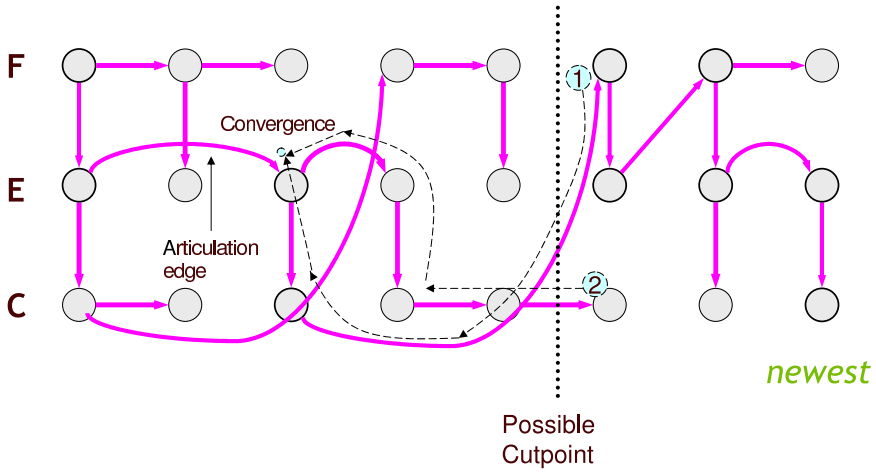
A second strategy for identifying articulation edges can be employed if the simpler strategy above does not yield articulation edges with sufficient frequency. It starts with finding cuts as above, but if there is more than one last-arriving edge in the cut-set, instead of just giving up, each edge is traversed backwards until a convergence point is found. The edge at the convergence point will be an articulation edge. Figure 5.2c illustrates the procedure. Since the critical path begins at the first node of the first instruction, it is guaranteed that a convergence will be found. For the strategy to be successful, convergence would need to be found relatively quickly, however, otherwise a large portion of the graph would need to be buffered in memory.



(a) A graph cut.



(b) Same as above but only showing last-arriving edges.



(c) More sophisticated articulation point algorithm.

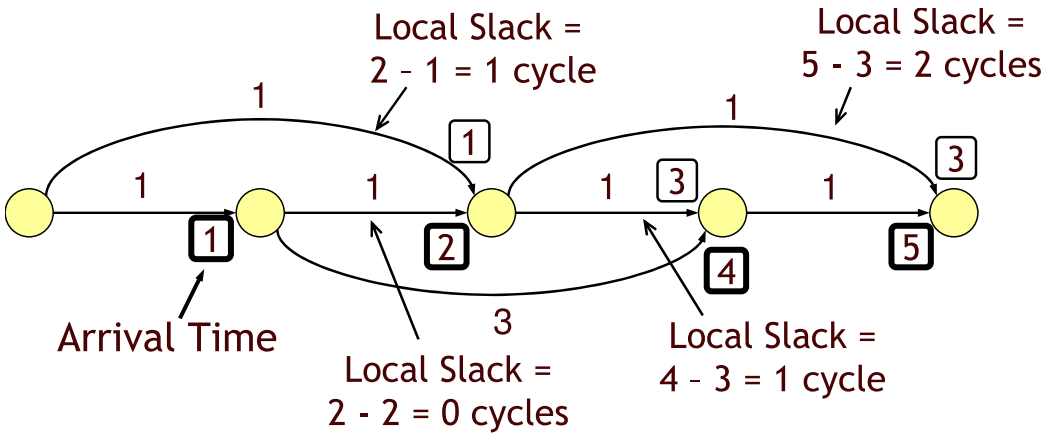
Figure 5.2: **Articulation edges aid in finding the critical path efficiently.**

In general, it is possible for no articulation edges to exist in the whole-program-execution graph. If this is the case, the critical path is not very dominant, in other words there are one or more secondary paths that are nearly as long as the critical one. To find the true critical path in this case, the entire graph would need to be buffered. For most applications, however, it is not that useful to identify the true critical path if it is not dominant, since other paths are nearly as long anyway. Any practical optimization would need to target both the critical and near-critical paths. Thus, in practice, it is usually effective to just mark edges in all of these paths as "critical".

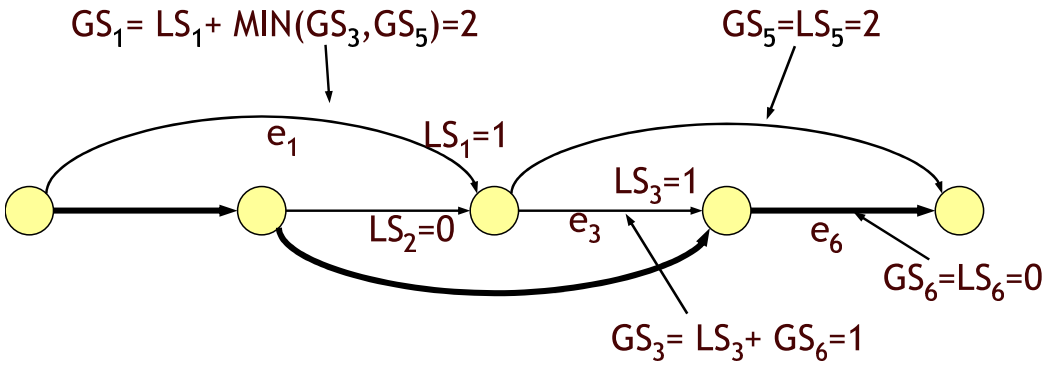
5.2 Computing Slack

Computing slack is a more difficult problem than criticality. This is not surprising since criticality can be inferred from slack values: any edge with zero slack is on the critical path. We will start the discussion of slack algorithms with the most basic metric (local slack) and use that as a basis for the more complex metrics (global and apportioned slack.) Note that there is a distinction between the slack of a node versus the slack of an edge, but the slack of a node can always be derived once we know the slack of all the edges, as will be made clear as we walk through the algorithms..

Local slack. The local slack of a *node* is determined by first computing the local slack of each *edge* in the graph. The local slack of an edge $e = u \rightarrow v$ is simply the number of cycles that the latency of e can be increased without delaying the target node v. The local slack of e is computed as the difference between the arrival time of the latest (*i.e.*, last-arriving) edge sinking on v and the arrival time of e (see Figure 5.3(a) for an example). The local slack of a node v is then the smallest local slack among the outgoing edges of v. Thus, the local slack of the middle node in the figure is $min(L_3, L_5) = 1$ cycle.



(a) Calculating local slack.



(b) Calculation global slack.

Figure 5.3: **Computing Local and Global Slack.**

```
ComputeGlobalSlack(G)
  // initialization
  arrival_time[n] ← 0 for all nodes n ∈ N[G]
  // forward pass
  for each node n ∈ V[G] in topological order
    for each incoming edge e to n
    if latency[e] + arrival_time[source[e]] > arrival_time[n]
      arrival_time[n] ← latency[e] + arrival_time[source[e]]
    // compute local slack
    for each incoming edge e to n
      local_slack[e] = arrival_time[n] − (latency[e] + arrival_time[source[e])
  // backward pass, to compute global slack
  global_slack[n] ← 0 for all nodes n ∈ N[G]
  for each node n ∈ N[G] in reverse topological order
    for each outgoing edge e from n
      if global_slack[e] < global_slack[n]
        global_slack[n] = global_slack[e]
    for each incoming edge e to n
      global_slack[e] = global_slack[u] + local_slack[e]
```

Figure 5.4: **Global slack algorithm.**

Global slack. As with local slack, we start by computing global slack of edges. The global slack of an edge e is the number of cycles that the latency of e can be increased without extending the graph's critical path. As with local slack, the global slack of a node v is the smallest global slack available among v's outgoing edges.

While local slack was computed by merely examining nodes and their edges, the computation of global slack involves backward propagation that accumulates local slack. Consider Figure 5.3(b) as an example. We start by knowing the value of local slack L_i of each edge e_i and end up computing, for each edge e_i, the value of global slack G_i for each edge.

In the example, G_3, the global slack of edge e_3, equals the sum of the local edge slacks L_3 and L_6. We can compute G_3 recursively, as the sum of L_3 and G_6. In general, the expression for computing the global slack of an edge e is $G_e = L_e + min(G_{out_1}, G_{out_2}, .., G_{out_n})$ where G_{out_1} to G_{out_n} are the global slacks of the outgoing edges of e's target node.

The complete algorithm is shown in Figure 5.4. While the algorithm is linear, there unfor-

tunately does not seem to be an equivalent to articulation edges when computing slack. For accurate slack computation, the graph of the entire program execution must be buffered (or saved to disk). We have found in practice, however, that global slack can be approximated with high precision by analyzing sufficiently large segments of the execution (a few tens of thousands of instructions), much smaller than the entire program run.

Apportioned slack. Having computed global slack, we are ready to compute apportioned slack. The goal of the algorithm is to apportion a certain amount of slack to *as many nodes as possible*, so that all nodes can be delayed (together) by the amount of slack apportioned to them without extending the critical path. The exact amount of slack we attempt to apportion to each node depends on the apportioning strategy.

The algorithm we use does not perform an optimal apportioning, but instead greedily apportions slack to the first nodes encountered during a forward pass, after computing global slack using the algorithm above. For example, assume an analyst wished to employ an apportioning strategy that gave five cycles of slack to as many nodes as possible. Performing this apportioning optimally is intractable, but our approach would provide a greedy solution that is (hopefully) good enough. As the forward pass encountered each node v, a check would determine whether enough global slack exists to apportion v five cycles of slack. If enough existed, v would be apportioned five cycles, and it would be ensured that no other nodes further downstream are apportioned those five cycles. This process would continue until the forward pass reaches the end of the program. The entire algorithm is shown in Figure 5.5.

```
ComputeApportionedSlack(G)
 for each node n ∈ N[G] in topological order
    available_slack[n] = global_slack[n]
    for each incoming edge e to n
    if available_slack[n] > available_slack[e] − local_slack[e]
       available_slack[n] = available_slack[e] − local_slack[e]
    Apportion slack to n up to available_slack[n] based on policy
    for each outgoing edge e from n
       available_slack[e] = available_slack[n] − apportioned_slack[u]
```

Figure 5.5: **Apportioned slack algorithm.**

5.3 Computing Cost

The most straightforward way to compute the $cost$ of an event, e, is to run two simulations: one with e idealized and one with no idealizations. Then, $cost(e)$ is simply the difference between the execution times of the non-idealized and idealized executions. Furthermore, $icosts$ can be computed similarly, since they are derived from simple cost measurements (*e.g.*, $icost(e_1, e_2) = cost(e_1, e_2) - cost(e_1) - cost(e_2)$).

In practice, though, this multiple-simulation approach is expensive, especially if the number of $cost$ measurements to be taken is very high. For instance, if a user wants to know the cost of every data cache miss, running so many simulations is undesirable. In other cases, a simulator may not even be available. Fortunately, we have developed algorithms to compute $cost$ much more efficiently using our dependence graph, which can be built from data collected by modified hardware performance counters.

The natural algorithm for computing the cost of an event on the graph would be to idealize the edge representing the event, and then measure the change in critical-path length this graph modification induces. While more efficient than running a full simulation, some analysis applications would require many critical-path length measurements, with a corresponding long analysis time.

In particular, if an analyst wished to measure the cost of every edge in the graph, the complexity would be quadratic, since each critical path measurement is $O(m)$, where m is the number of edges. We employ some optimizations that reduce this complexity in practice (see Section 5.4 below), but before we get to those, we will first discuss a special-purpose algorithm that can compute the cost of every single edge in the graph in near linear time.

Near-linear-time algorithm. The key observation to the algorithm is to realize the cost of an edge on the critical path is equal to the the slack of the second-most critical path. In other words, the cost of a critical edge e is equal to the minimum of the slacks of all edges parallel to e. The intuition behind this is that, since the cost of an edge is the reduction in critical-path length realized when the edge is idealized, the maximum reduction possible is the difference between the current critical-path length and the length of the second-most critical path. This value is precisely the global slack of the second-most critical path.

This observation alone is not enough to yield a linear time algorithm, however. In general, finding the set of edges parallel to an edge e requires $O(m)$ time, where m is the number of edges. Thus, to find such sets for every possible edge, the straightforward algorithm would require $O(m^2)$ time. Careful bookkeeping is required to reduce this complexity.

The first step is to topologically sort the critical-path nodes and assign each a number k such that if node n_1 is assigned k_1 and n_2 is assigned k_2 and $k_1 < k_2$, n_1 is an ancestor of n_2 (see Figure 5.6). Then, for each non-critical edge e, a range is found $D = (k_b, k_e)$ where every edge on the critical path between k_b and k_e is parallel to e. These ranges can be found with two passes of a simple dataflow analysis (see Figure 5.7). Finally, the critical-path is traversed in a forward direction, maintaining a priority queue of all edges parallel to each critical-path edge. The cost of

```
AssignNumbers(G, S)
 CURID ← 0
 ID(N) ← ∅ for all N
 ENQUEUE(CQ, S)   while CQ not empty or NCQ not empty
   if NCQ not empty
     N ← DEQUEUE(NCQ)
   else
     N ← DEQUEUE(CQ)
   ID(N) ← CURID
   CURID ← CURID + 1
   for N_c in CHILDREN(N)
     if ID(N_c) = ∅
       if N_c not critical
         ENQUEUE(NCQ, N_c)
       else
         ENQUEUE(CQ, N_c)
```

Figure 5.6: **Algorithm to topologically sort and assign numbers to nodes as part of an algorithm to compute the cost of every node.** S is the first (starting) node of the graph. CQ is a queue for holding critical nodes. NCQ is a queue for holding noncritical nodes. The resulting topological order is represented by the numbers stored in $ID(N)$ for each node N in the graph.

each critical edge is the non-critical edge in the priority queue with the least global slack.

The complexity of the algorithm is $O(mlogn)$ since the priority queue implementation requires at most $O(logn)$ complexity for each critical edge. In practice, however, the number of edges parallel to a critical-path edge will be bounded by hardware resource constraints, *e.g.*, due to a finite-sized reorder buffer. Thus, the complexity is effectively linear.

Note that the algorithm presented computes the cost of every *single* edge in the graph. In other words, the algorithm does not compute the aggregate cost of two or more edges, which is necessary for determining interaction costs. It seems that computing the cost of every pair of edges is more difficult than for every single edge, probably requiring an algorithm of quadratic complexity.

In practice there are a some mitigating conditions, however. Most importantly, it is not interesting to compute the cost of every pair of edges since it is easily determined that some edges are too distant from each other to have any interaction. So, the quadratic complexity is only within

```
FindExtent(G)
  for each node N of G in topological order
  if N is critical
     EXTENT(N) ← ID(N)
  else
     k ← minimum EXTENT(j) of all nodes j that are parents (immediate ancestors) of n
     EXTENT(N) ← k
  return EXTENT(n)

FindRanges(G)
  BEGIN(N) ← FindExtent(G)
  Let G' be G with all edges reversed
  END(N) ← FindExtent(G')
```

Figure 5.7: **Algorithm to find ranges of critical nodes parallel to each noncritical node** $BEGIN(N)$ and $END(N)$ contain the beginning and ending identifiers for the range for each node N.

a local region of the graph.

5.4 Algorithms for Dynamic Graphs

The algorithms for slack and cost discussed above make an important assumption concerning the graphs that they are operating on: that they are static (unchanging) during the analysis. For most analyses, this is an acceptable assumption. It is sometimes useful, however, to modify the graph while the processing is occurring. For instance, if we are using the graph to determine good cutpoints in which to divide a program into multiple threads (an application sketched in a later chapter), inter-processor latencies would be placed on different edges for each possible cutpoint. Such an application is going to necessarily require a more time-consuming analysis, but by taking advantage of the unique structure of a microexecution graph, the analysis can be done in a reasonably tractable way.

The basic approach we take is to measure slack and cost using the straightforward approach of computing critical-path lengths, but instead of measuring the critical path of the entire

```
ComputeSlack(G, E_m, localSlack)
 // initialization
 count ← 0
 lastSlackNode ← ∅
 for each n ∈ N
    slackNode[n] ← ∞
 for each e_m ∈ E_m
    slackEdge[e_m] ← localSlack[e_m]
 // do propagation
 for each n ∈ N in program order (starting at earliest targetNode[e_m ∈ E_m]) {
    for each e_i ∈ IncomingEdges[n] − E_m
       slackEdge[e_i] ← slackNode[sourceNode[e_i]] + localSlack[e_i]
    slackNode[n] ← MIN(slackEdge[e_i ∈ IncomingEdges[n]])
    // check condition for ending propagation
    if slackNode[n] = lastSlackNode
       count ← count + 1
    else {
       lastSlackNode ← slackNode[n]
       count ← 0
    }
    if count ⩾ ROB_SIZE
       return slackNode[n]
 }
```

Figure 5.8: **Algorithm to compute slack of an individual event.** Note, due to the structure of our model, this single event could occur on multiple edges. For instance, the effect of a cache miss could occur on multiple EE edges as well as an EC edge. We could re-structure the model to avoid such difficulties, if desired.

microexecution for each measurement, we introduce a *convergence* condition such that the entire graph need not be examined. The convergence condition is based on specific knowledge of how the graph is structured, specifically that parallelism is limited to the span of the reorder buffer. Any dynamic alterations to the graph will be done individually for each slack or cost measurement, *i.e.*, each measurement in essence has its own graph. Note that since propagation only proceeds a limited amount before convergence is reached, we avoid constructing a copy of the *entire* graph for each measurement. The algorithms are shown in Figure 5.8.

Both algorithms accept three inputs. $G = (N, E)$ is the graph of the microexecution and contains a set of nodes N and a set of edges E. The function $IncomingEdges[n]$ returns the set of incoming edges into node n; function $targetNode[e]$ and $sourceNode[e]$ return, respectively,

```
ComputeCost(G, E_m, localSlack)
 // initialization
 count ← 0
 lastCostNode ← ∅
 for each n ∈ N
    costNode[n] ← 0
 for each e_m ∈ E_m
    costEdge[e_m] ← ∞
 // do propagation
 for each n ∈ N in program order (starting at earliest targetNode[e_m ∈ E_m]) {
    for each e_i ∈ IncomingEdges[n] − E_m
       costEdge[e_i] ← costNode[sourceNode[e_i]] + localSlack[e_i]
    costNode[n] ← MIN(costEdge[e_i ∈ IncomingEdges[n]])
    // check condition for ending propagation
    if costNode[n] = lastCostNode
       count ← count + 1
    else {
       lastCostNode ← costNode[n]
       count ← 0
    }
    if count ⩾ ROB_SIZE
       return costNode[n]
 }
```

Figure 5.9: **Algorithm to compute cost of an individual event.**

the target and source nodes of edge e. The input E_m is the set of edges that are to be measured for slack or cost. The final input $localSlack$ is an array containing the local slack of each edge in the graph. Local slack of an edge e incoming to a node n is defined as $LA\_EdgeCycle[n] - EdgeCycle[e]$ where $LA\_EdgeCycle[n]$ returns the cycle the last arriving edge into n is resolved and $EdgeCycle[e]$ returns the cycle edge e was resolved.

The algorithms are very similar except for initialization. In the slack algorithm, we start with a small value of slack (just the local slack at the edge being measured, E_m) and continually refine this value as we observe more slack on edges in paths leading from E_m. Conceptually, we can think of the algorithm as searching for the path from E_m to the critical path that has the *least* cumulative slack. This cumulative slack is reported as the slack for E_m.

In the cost algorithm, we know that E_m must be on the critical path (else it has a slack

rather than a cost.) We start with a large value of cost, ∞, and continually refine this value as we discover paths that are closer and closer to the critical path. Conceptually, we are trying to determine how far E_m is from the second-most critical path.

The convergence condition is the same for both algorithms. If the same slack or cost value has been propagated across every edge for the last ROB_size instructions, we know that the value will never change.

5.5 Summary

In this chapter, we discussed the graph algorithms that operate on a dependence graph of the microexecution to extract the metrics of interest for our criticality analysis (*i.e.*, the critical path, slack, cost, and interaction cost). These algorithms are used in two ways in our work. The first is to implement them directly in a simulator to help computer architects better understand the performance characteristics of the machines they are building. Secondly, the algorithms form the basis for some of the hardware techniques and profiling infrastructures discussed in the next chapter.

Chapter 6

Hardware Support

For many applications of our performance metrics, we want to perform measurements of real programs executing on real machines. To make this possible, we provide hardware support for measuring criticality, slack, and interaction cost. In the next two sections, we describe our solutions for detecting criticality and slack in hardware. Then, in Section 6.3, we describe an alteration to traditional performance counter infrastructure that enables us to build full-featured graphs from scant information collected during a program's execution.

6.1 Criticality Analyzer and Predictor

Recall the observation of the previous chapter that the critical path consists of only last-arriving edges (see Section 5.1). For the hardware criticality analyzer, we take this observation a step further: for the goal of detecting criticality, the portion of the graph consisting of non-lastarriving edges does not even need to be constructed. Furthermore, since the critical path consists of the chain(s) of last-arriving edges spanning from the beginning of the program to the end, it can be

target node	edge	last-arriving condition
F	$E_{i-1} \rightarrow F_i$	if i is the first committed instruction since a mispredicted branch.
	$C_{i-w} \rightarrow F_i$	if the re-order buffer was stalled the previous cycle.
	$F_{i-1} \rightarrow F_i$	if neither *EF* nor *CF* arrived last.
E	$F_{i-1} \rightarrow E_i$	if all the operands for instruction i are ready by the time i is dispatched.
	$E_j \rightarrow E_i$	if the value produced by instruction j is the last-arriving operand of i and the operand arrives after instruction i has been dispatched.
C	$E_i \rightarrow C_i$	if instruction i delays the in-order commit pointer (*e.g.,*, the instruction is at the head of the re-order buffer but has not completed execution and, hence, cannot commit).
	$C_{i-1} \rightarrow C_i$	if edge *EC* does not arrive last (*i.e.,*, instruction i was ready to commit before in-order commit pointer permitted it to commit).

Table 6.1: **Determining last-arriving edges.** Edges are grouped by their target node. Every node must have at least one incoming last-arriving edge. However, some nodes may not have an outgoing last-arriving edge. Such nodes are non-critical.

found without knowing the operation latencies associated with the edges. (If more than one such chain of last-arriving edges exist, each chain is equally critical.) So, if we can find a simple way of identifying last-arriving edges, it may not be necessary for hardware to measure latencies at all.

Fortunately, it is possible to identify last-arriving edges in hardware using simple rules. A few examples: for execution edges, the issue logic must already detect when a dependence is the last-arriving one, since that is the cue for the dependent operation to issue. For fetch nodes, monitoring processor events is enough: *e.g.*, if a branch misprediction prohibits a fetch from occurring, the misprediction is the last-arriving edge. A summary of all the rules for a three node model are shown in Figure 6.1. These rules are all the hardware needs to determine last-arriving edges and, hence, the critical path.

Nonetheless, even with this dramatic reduction in work required for creating the graph, the algorithm is still very expensive for a hardware implementation. The reason is that a relatively large portion of the graph (between articulation edges) needs to be buffered before the critical-path can be found via backwards traversal of last-arriving edges (see Section 5.1). This involves a large

amount of storage and control logic within a processor.

We solve this problem by transforming the *backwards* traversal into a *forward* one. We do this by employing the following property: if there exists a chain of last-arriving edges from a node N to the end of the program, N is critical. We know this because if N is delayed by any amount (*e.g.,* one cycle), the execution time of the entire program will necessarily be increased (due to the lack of any slack). So, if a forward traversal along last-arriving edges from N reaches the end of the program, we know that N is critical. On the other hand, if there is no such chain, N cannot be critical, since it has some (global) slack. Below we describe the algorithm we use in detail. Note that, in order to gain criticality information while the program is still running, we employ an approximation that could, potentially, lead to some instructions being falsely identified as critical.

Token-passing Algorithm. The complete token-passing training algorithm is shown in Figure 6.1. It works through frequent *sampling* of the criticality of individual nodes of instructions. To take a criticality sample of node n, a token is planted into n (**step 1**) and propagated *forward* along *all* last-arriving edges (**step 2**). If there is more than one outgoing last-arriving edge the token is replicated. At some nodes, there may be no outgoing last-arriving edges for the token to propagate further. If all copies of the token reach such nodes, the token *dies*, indicating that node *n* **must not** be on the critical path, as there is definitely no chain of last-arriving edges from the beginning of the program to the end that contains node n. On the other hand, if a token remains alive and continues to propagate, it is **increasingly likely** that node *n* is on the critical path.

This is the point where our approximation comes in. Instead of propagating the token all the way to the end of the program, we stop after the processor has committed some threshold number of instructions (called the *token-propagation-distance*). At this point, we check if the token

1. Plant token at node *n*.
2. Propagate token forward along *last-arriving* edges.
   If a node does not have an outgoing last-arriving edge,
   the token is not propagated (*i.e.,*, it *dies*.)
3. After allowing token to propagate for some time,
   check if the token is still alive.
4. If token is alive, train node *n* as critical;
   otherwise, train *n* as non-critical.

Figure 6.1: **The token-passing training algorithm.**

is still alive (**step 3**). If it is, we assume that node *n* was critical; otherwise, we know that node *n* was non-critical. The larger the token-propagation-distance, the less likely any instructions will be falsely identified as critical (the token analyzer never, as regards to the graph, incorrectly identifies an instruction as non-critical).

The result of the token propagation is then used to train the predictor (**step 4**). More will be said about the predictor in Section 6.1.2.

Token-passing analyzer parameters discussion. There are several parameters that are important to consider in the design of the analyzer. One of the most important is the $token\_propagation\_distance$, which is the number of instructions that must commit during a token's lifetime in order for the node in which the token was planted to be considered critical (**step 3** of Figure 6.1). The larger the $token\_propagation\_distance$, the more accurate the analyzer will be in detecting criticality. On the other hand, more nodes will be sampled if the $token\_propagation\_distance$ is smaller, since the token will be available to be replanted more quickly. In addition, the criticality information for a node is obviously not available until after the detection completes, which may cause us to miss out on optimization opportunities if one or more dynamic instances of the static instruction being sampled are fetched and executed while waiting for $token\_propagation\_distance$ instructions to commit.

One way to get the benefits of a large $token\_propagation\_distance$ while maintaining a high sampling rate is to increase the number of tokens. This comes at the cost of replicating the token-array used for storing and propagating a token. As we will see in the next section, however, the cost of implementing the token-passing analyzer is so low that such replication is relatively inexpensive.

A final parameter of the analyzer is the policy used to decide which nodes to plant tokens into (**step 1**), which determines how many samples would be taken of each static instruction. Experimentally, we have found that this policy does not matter much in practice. If tokens were planted in a completely deterministic manner, however (*e.g.*, immediately after they are freed), there are pathological cases that would lead to many static instructions never being sampled. For this reason, we use a randomized policy.

6.1.1 Hardware Implementation of the Analyzer

We'll discuss two approaches to implementing the token-passing algorithm in hardware. The first performs the token propagation via read-modify-write operations on a small array. The benefit of this approach is that (except for the last-arriving edge detection) all of the logic for the token-passing is located outside any datapaths of the processor. The only information that must be provided by the processor are the last-arriving edges. One disadvantage is that the bandwidth required to transmit the last-arriving edges from the processor core to the analyzer is fairly high. As an estimate, the number of bits the must be transmitted each cycle would be equal to *number of instructions committed per cycle* $\times$ *number of nodes per instruction* $\times$ *number of bits to represent the origin of a last-arriving edge*. For a 6-wide machine with 3 nodes per instruction and a 256 instruction ROB, the expression yields $6 \times 3 \times 8 = 124$ *bits*. More importantly, the wires for this

information will be coming from all over the chip, which can lead to routing problems. Our alternative implementation performs the token-passing in a distributed manner, local to where each of the last-arriving edges are determined. This approach eliminates the read-modify-write operations and the large number of wires coming out of the processor at the expense of some modularity, since the analyzer is no longer centralized.

Centralized (Off-the-core) Implementation

Our centralized implementation performs the token passing outside of the main processing core by performing read-modify-write operations to a small array (Figure 6.2). The array stores information about the segment of the dependence graph for the *ROB_size* most recent instructions committed. One bit is stored for each node of these instructions, indicating whether the token was propagated into that node. Note that the array does not encode any dependence edges; their effect is implemented by the propagation step (see step 2 below).

As each instruction commits, it is allocated an entry in the array, replacing the oldest instruction in a FIFO fashion. A token is planted into a node of the instruction by setting a bit in the newly allocated entry (**step 1** of Figure 6.1).

To perform the token propagation (**step 2**), the processor core provides, for each committing instruction, identification of the source nodes of the last-arriving edges targeting the three nodes of the committing instruction. (An identifier for a last-arriving edge is simply the instruction number assigned to the instruction containing the source node, along with the node type. The instruction numbers are assigned in program order in the range of 0 to ROB_size, wrapping when the maximum extent is reached.) For each source node, its entry in the token array is read (using its identifier as the index) and then written into the target node in the committing instruction. This

simple operation achieves the desired propagation effect. Finally, note that the reason why the token array does not need more than *ROB_size* entries is the observation that no critical-path dependence can span more than *ROB_size* instructions.

Checking if the token is still alive (**step 3**) can be easily implemented without a scan of the array, by monitoring whether any instruction committed in the recent past has written (and therefore propagated) the token. If the token has not been propagated in the last *ROB_size* committed instructions, it can be deduced that none of the nodes in the token array holds the token, and, hence, the token is *not* alive. Finally, based on the result of the liveness check, the instruction where the token was planted is trained (**step 4**) by writing into the critical-path prediction table, using the hysteresis-based training rules in Table 6.2.

After the liveness check, the token is *freed* and can be re-planted (**step 1**) and propagated again. The token planting strategy is a design parameter that should be tuned to avoid repeatedly sampling some nodes while rarely sampling others. In our design, we chose to randomly re-plant the token in one of the next 10 instructions after it is freed.

Hardware Costs. Now we will analyze the hardware expense of the token-passing array in the centralized implementation. As mentioned above, the frequency of the sampling is influenced by both the token-propagation-distance and the number of tokens available for planting. In this implementation, additional tokens increase the size and number of ports required of the token array; but they are inexpensive in terms of additional control logic since all of the tokens can be read and written together during propagation. For the propagation distance we chose (500 + ROB size = 1012 dynamic instructions), eight simultaneous in-flight tokens was sufficient. For this configuration, the token array size is 1.5 kilobytes (reorder buffer size $\times$ nodes $\times$ tokens $= 512 \times 3 \times 8$ bits).

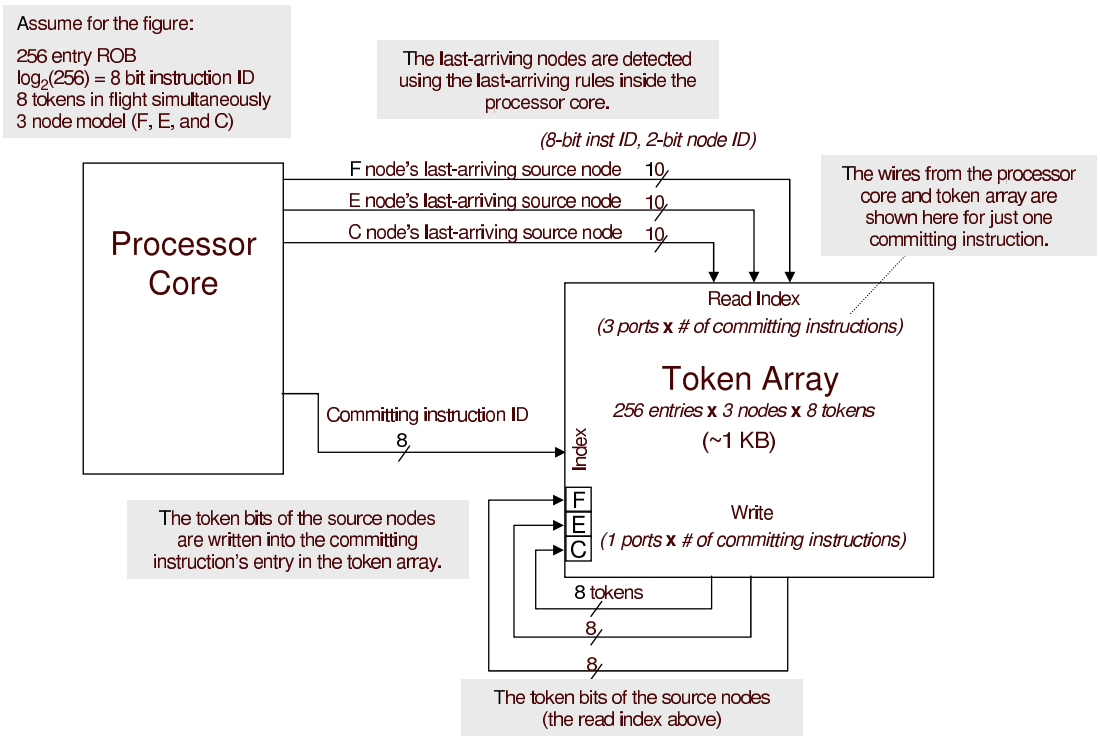


Figure 6.2: **Training path of the critical-path predictor.** Training the token-passing predictor involves reading and writing a small (less than one kilobyte) array. The implementation shown permits the simultaneous propagation of eight tokens.

Although the number of ports of the token array is proportional to the maximum commit bandwidth (as well as to the number of simultaneous last-arriving edges), due to its small size, the array may be feasible to implement using multi-ported cells and replication. Alternatively, it may be designed for the average bandwidth. Bursty periods could be handled by buffering or dropping the tokens.

Distributed (Throughout-the-core) Implementation

The key to the distributed implementation is that the tokens are attached as extra control bits to each instruction as it flows through the pipeline. There would be one bit for each node, and multiple sets of these bits if multiple tokens are supported. Although it may sound expensive to

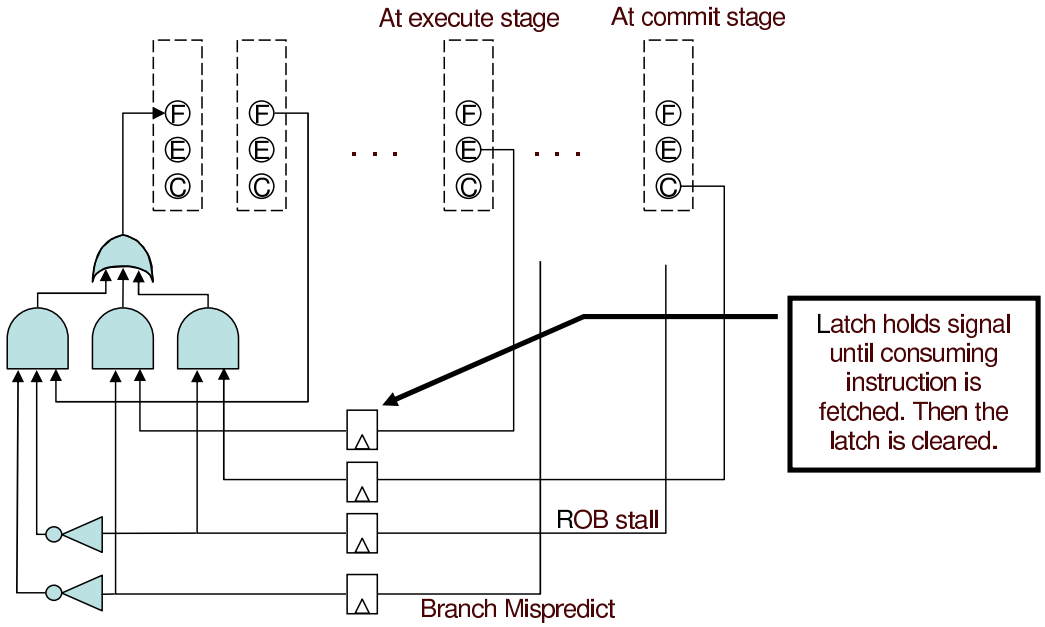


Figure 6.3: **Example of token passing in distributed criticality analyzer implementation.** The logic for passing a token into the D-node of an instruction being dispatched is shown. Logic for the other nodes would be similar in flavor.

attach a few bits to every dynamic instruction as they flow through the machine, most processor implementations transmit hundreds of bits for each instruction already. A few extra bits represents rather small overhead.

In this style of implementation tokens are planted (**step 1**) into a node of an instruction by setting the node's token bit as the instruction is fetched (or at least before the corresponding stage of the pipeline).

Token propagation (**step 2**) is performed "inline" as the instruction flows through the pipeline. By inline we mean that the token-passing logic resides within the core of the machine and that the token is passed as soon as the last-arriving edges are detected. While at first this may sound intrusive to the operation of the processor core, the propagation operation is exceedingly simple, perhaps even simpler than recording the last-arriving edges for later propagation in the centralized

scheme. When one of the last-arriving rules is observed, instead of recording the minimal information to communicate to a token-passing backend as in the centralized scheme, a single bit is set in the consuming instruction.

For example, consider the token-passing logic for tokens flowing into the F node of instruction i, illustrated in Figure 6.3. If an ROB stall occurred such that the fetch of i was delayed, the value of the C node at the instruction that caused the stall is written into the bit representing i's F node. On the other hand, if i is the first correct-path instruction after a branch misprediction, the value of the E node of the mispredicted branch is written into i's F node. Finally, if neither of the other conditions are met, the value of the previous fetched instruction's ($i-1$'s) F node is written into i's F node.

Checking if the token is still alive (**step 3**) is performed the same way as in the centralized scheme. The token bits attached to the instructions are observed as each instruction commits. If none of the bits for a token are set for the last *ROB_size* instructions, we can conclude the token is dead. After the token dies (or the token propagation distance is exhausted), the token can be "freed" and replanted (**step 1**).

Hardware Cost. When considering the design parameters of the token-passing analyzer, there is one significant difference as far as incremental hardware expense between the centralized and distributed implementations: the cost of additional tokens to increase sampling rate. In the distributed implementation, increasing the number of tokens incurs a cost of extra bits attached to each instruction as it flows through the pipeline. This expense, since it occupies precious real estate within the processor core, suggests that extra tokens are more expensive than in the centralized scheme.

6.1.2 History-based Prediction

In the previous section, we discussed implementations of the criticality *analyzer*, which determines with a high degree of accuracy the criticality of a single dynamic instruction (or, more specifically, micro-operation). Due to token propagation latency, however, the analyzer does not return a criticality result until long after the instruction has completed execution, which is far too late for applying most optimizations. For example, a criticality-based instruction scheduling policy would obviously need to know which instructions are critical before the instructions are scheduled for execution.

Our solution is to use the analyzer to train a critical-path table, which is indexed by the PC of the instruction. As another dynamic instance of a previously analyzed static instruction is fetched, a prediction of criticality is retrieved from the table. Then, this information can be used to make optimization decisions.

For this type of prediction scheme to work, the criticality of instructions must exhibit "locality", in the sense that different dynamic instances of the same static instruction have similar criticality characteristics. Of course, this will not always be true. For example, a branch instruction and the instructions it depends upon are likely to be non-critical when the branch is predicted correctly and critical when it is predicted incorrectly. Our hope is that some static instructions are much more likely to be critical than other instructions. The goal of our hardware analyzers, then, would be to identify this more-likely-to-be-critical set of instructions.

Since the criticality of different nodes (fetch, execute, commit) have very different characteristics, we will discuss each individually. Figure 6.4 shows a characterization of criticality for the execute (E node) of instructions. From the figure we see that very few instructions are critical

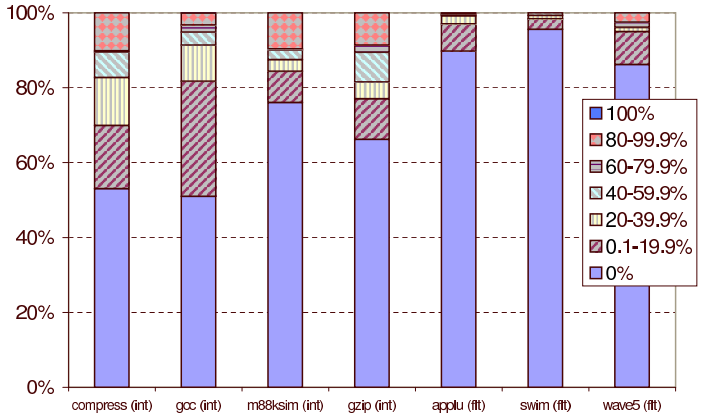
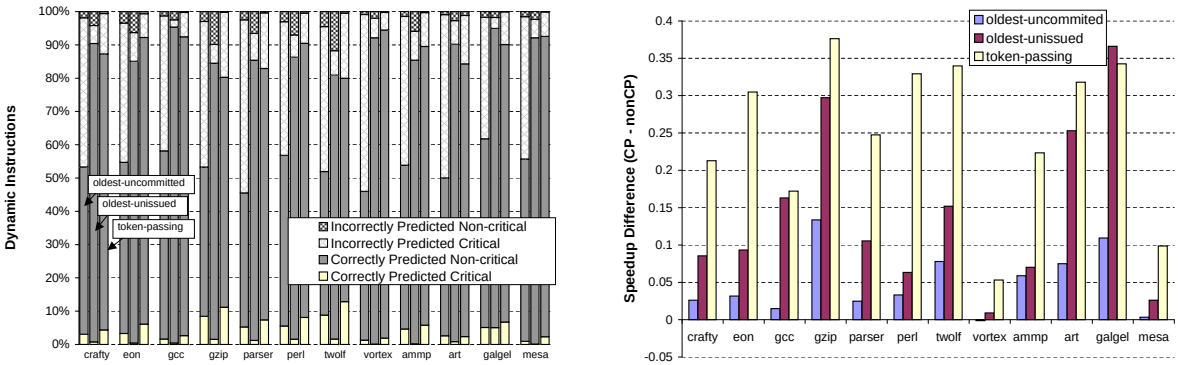


Figure 6.4: **Dynamic to Static Histogram.** For each static instruction, the percentage of its dynamic instances that are critical (its "criticality frequency") was recorded. The figure shows the percent of static instructions that had a criticality frequency within each range specified in the legend. The y-axis is the percent of static instructions, *weighted* by their dynamic frequency.

all the time. At first glance, this seems to be bad news for a history-based predictor, since few static instructions can be identified as always critical. There are, however, many instructions which are never critical, and it is certainly true that some static instructions are more often critical than others. The goal of our predictor, then, will not to predict the precise criticality of every dynamic instruction but instead to identify those static instructions which exhibit criticality more frequently than others.

Critical path prediction table	12 kilobytes (16K entries * 6 bit hysteresis)
Token propagation Distance	1012 dynamic instructions (500 + ROB size)
Maximum number of Tokens in flight simultaneously	8
Hysteresis	Saturate at 63, increment by 8 when training critical, decrement by one when training non-critical. Instruction is predicted critical if hysteresis is above 8.
Planting Tokens	A Token is planted randomly in the next 10 instructions after it becomes available.

Table 6.2: **Configuration of token-passing predictor.**



(a) Comparison against "ideal" CP, computed offline. **(b)** Comparison via latency reduction.

Figure 6.5: **The token-passing predictor is very successful at identifying critical instructions.** (a) Comparison of the token-passing and two heuristics-based predictors to the "ideal" trace of the critical path, computed according to the model from Section 3.2.2. The token-passing predictor is over 80% (88% on average) accurate across all benchmarks and typically better than the heuristics, especially at correctly predicting nearly all critical instructions. (b) Plot of the difference of the performance improvement from decreasing critical latencies minus the improvement from decreasing non-critical latencies. Except for *galgel*, the token-passing predictor is clearly more effective.

Consider the number of samples required to obtain a good measure of criticality for a static instruction. Remember we are attempting to identify the set of static instructions that are most likely to be critical. If instructions in that set are critical during, say, one-fourth of their dynamic instances, on average of four samples will need to be taken to detect its criticality.

We have found that this goal is best achieved by using a predictor that has hysteresis biased towards predicting that an instruction is critical. In other words, a static instruction is quickly learned to be critical when one of its dynamic instances is found to be critical, while many of its dynamic instances must be detected noncritical before the static instruction is considered noncritical. Empirically, we found the scheme described in Table 6.2 works well.

Predictor Accuracy

The two most meaningful measures of accuracy of the predictor for dynamic hardware optimizations are (1) what fraction of dynamic instructions that are critical are predicted as critical?

and (2) what fraction of noncritical dynamic instructions are predicted noncritical? Both questions are interesting, since a typical optimization using criticality, *e.g.,* resource arbitration, would perform best if all critical instructions are given high priority *and* all noncritical instructions are given lower priority.

Recall that the locality measurements of the previous section showed that few static instructions are critical for a majority of their dynamic executions. That fact combined with our design decision to identify those static instructions that are more likely than others to be critical (even if they are only occasionally critical) causes us to expect to see a large number of noncritical dynamic instructions predicted as critical. From Figure 6.5, however, we are pleasantly surprised to find only approximately 10% of instructions are incorrectly predicted critical. In addition, very few dynamic critical instructions are predicted noncritical (less than 2%). All in all, only 15% of the instructions are predicted critical and that 15% includes nearly all of the actually critical instructions. This result speaks well for using the criticality predictor for resource arbitration and policy decisions.

Comparison to Heuristics-based Approaches. Our token-passing predictor is designed using a *global* view of the critical path. An alternative is to use *local* heuristics that observe the machine and train an instruction as critical if it exhibits a potentially harmful behavior (*e.g.,*, when it stalls the reorder buffer). A potential advantage of a heuristic-based predictor is that its implementation could be trivially simple.

Our evaluation suggest that heuristics are much less effective than a model-based predictor. We compare our predictor to two heuristic predictor designs of the style used in Tune, *et al.* [110]. The first predictor marks in each cycle the oldest *uncommitted* instruction as critical. The second predictor marks in each cycle the oldest *unissued* instruction if it is not ready to issue. We

used the hysteresis strategy presented in their paper.

We first compare the three predictors to the "trace" of the critical path computed by the simulator using our model from Section 3.2.2. (The trace is guaranteed to accurately identify critical or noncritical instructions to the extent that they can be correctly identified using the dependence graph.) The results, shown in Figure 6.5(a), show that we predict more than 80% of dynamic instructions correctly (both critical and non-critical) in all benchmarks (88% on average). Our predictor does a better job of correctly predicting *critical* instructions than either of the two heuristics-based predictors. Note that the *oldest-unissued* predictor has a relatively low misprediction rate, but tends to miss many critical instructions, which could significantly affect its optimization potential.

Second, to perform a end-to-end comparison that factors out our critical-path model, we study the effectiveness of the various predictors with the same experiment that we used for validating the model—extending all latencies by one cycle and then decreasing critical and non-critical latencies. For an informative comparison, we plot the difference of the performance improvement from decreasing critical latencies minus the improvement obtained when decreasing non-critical latencies. This yields a metric of how good the predictor is at identifying performance-critical instructions. The larger the difference, the better the predictions. The results are shown in Figure 6.5(b). The token-passing predictor typically outperforms either of the heuristics, often by a wide margin. Also, notice that the heuristics-based predictors are ineffective on some benchmarks, such as *oldest-uncommitted* on *gcc* and *mesa* and both *oldest-uncommitted* and *oldest-unissued* on *vortex*. While a heuristic could be devised to work well for one benchmark or even a set of benchmarks, explicitly modeling the critical path has the significant advantage of robust performance over a variety of workloads.

6.2 Slack Analyzer

On the surface, slack seems much more difficult to analyze in hardware than criticality. After all, the software algorithm we use requires us to measure latencies on the edges and perform two passes over the entire graph. There is no obvious way of determining slack using only last-arriving edges or only one forward pass.

We simplify the problem considerably, however, with a trick that effectively reduces the problem of slack computation to one of criticality. Remember that the slack of an event is the number of cycles that event can be delayed without increasing execution time. So, if we delayed an event by n cycles, but the execution time remained unchanged, we could conclude that the event has at least n cycles of slack. Determining whether execution time increases due to a delay is very difficult, however, since execution time is not generally known until the program completes.

Our solution uses the criticality of an event as an indication of whether execution time increased. From the definition of criticality, an event that is not on the critical path has no effect on execution time. So, if the event is non-critical after a delay of n cycles, the event must have at least $n+1$ cycles of slack. (The event has at least $n+1$ cycles of slack since slack is defined by the delay that can be incurred without *increasing* the length of the critical path, as opposed to simply making the event critical.)

The hardware algorithm, thus, answers the question "does the dynamic micro-operation event e represented by node n have k cycles of slack?", using the procedure of Figure 6.6.

Notice that the procedure does not detect the precise amount of slack a micro-operation has. Our approach is to obtain an approximate, averaged value of slack for a *static* instruction by repeatedly applying the above procedure with different delays to many of its dynamic instances.

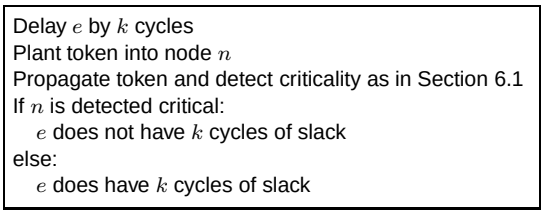


Figure 6.6: **Algorithm for measuring slack in hardware.**

This approach has the implicit requirement that there is a *locality of slack*, meaning that different dynamic instances of a static instruction have similar amounts of slack. (Otherwise the algorithm would be searching for a single slack value that does not exist.) We will discuss the characteristics of slack locality that we discovered in our benchmarks below.

6.2.1 Locality of Slack

From the locality of criticality experiments above, we found that many static instructions are always noncritical across all (or the vast majority) of their dynamic instances. That data did not tell us, however, whether the dynamic instances of those static instructions each have the same amount of slack. In this section we perform locality experiments to test whether the implicit analyzer above can be converted into a history-based predictor.

Our experiments present good news: 68% of static instructions (dynamically weighted) *almost always* have enough slack to double their latency (precisely, they have enough slack on at least 90% of their dynamic instances; see Figure 6.7). More significantly, this slack represents about 80% of all apportioned slack (that is, 80% of slack exploitable by an oracle predictor that correctly predicts the slack of every dynamic instruction).

The methodology we used is as follows. First, we computed the apportioned slack using one of the many possible optimization-specific apportioning polices (*e.g.*, the latency-plus-one strat-

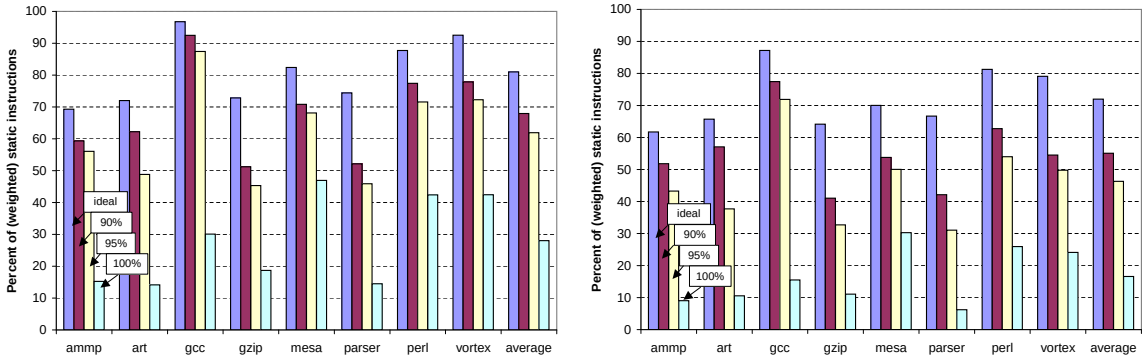


Figure 6.7: **Mapping dynamic slack behavior to static instructions.** Uses latency-plus-one-cycle apportioning. On the y-axis, the number of slackful static instructions is weighted by the number of each static instruction's dynamic instances.

egy introduced in Section 4.3.2). Next, we identified *slackful* static instructions. A static instruction is slackful if D% of its dynamic instructions contained apportioned slack, where D was varied from 90 to 100.

Figure 6.7a plots the amount of slackful static instructions for the latency-plus-one apportioning strategy. The chart also plots the total amount of apportioned slack (labeled *ideal*). This slack could be exploited with an oracle predictor that is correct on each dynamic instruction. Note that while relatively few static instructions are slackful all the time (28%, on average), allowing just 5% "misprediction rate" (*i.e.*, requiring them to be slackful 95% of the time) brings this amount to 62%, on average.

As a second example, Figure 6.7b plots the same data using the five-cycle apportioning policy, which attempts to apportion five cycles of slack to as many instructions as possible (see Section 4.3.2). From this chart, we see that slightly less than half of the static instructions can be apportioned five cycles of slack with a low 5% misprediction rate.

Sampling Requirements. Now consider the number of samples required to measure the average slack of a static instruction. If we could assume every dynamic instance of a static instruction had the same slack, a binary search would arrive at the correct value in $log_2(Max\_Slack)$ steps, where Max_Slack is the largest value of slack that would be explored. So, if we didn't care about precise values of slack over 128 cycles, $log_2(128) = 7$ samples would be required.

Unfortunately, slack does vary substantially from one dynamic instance of a static instruction to the next, according to the microarchitectural behavior surrounding the dynamic instruction's execution. Not only does this variance increase the number of samples required to obtain an accurate reading, it also complicates the binary search used to converge on an average value. In our experimentation, we have found that obtaining precise values of slack using this "delay-and-observe" approach is very difficult, possibly intractable.

Fortunately, precise values of slack are not usually needed to make effective use of slack information in optimizations. Instead, we are generally concerned whether an instruction has *enough* slack, where "enough" is defined by the specific application that the analyst is interested in.

Besides this, the types of machines for which slack is especially useful may not provide the capability to measure slack independent of its heterogeneous resources. For example, on a machine with both fast and slow functional units, an instruction will appear to have different slack, depending on which functional units it (and its dependents) were executing when its slack was sampled. Below we discuss a predictor design that takes these realities into account.

6.2.2 Implicit-Slack Predictor

We call the predictor described above that attempts to arrive at the precise value of slack an *explicit* slack predictor. Here we describe an alternative which, instead of arriving at a precise value, produces a slack-based categorization of instructions according to the optimization or heterogeneous resources being employed. We call this style of slack prediction *implicit*, since an instruction is known to have slack due to its ability to be delayed by some slow resource without increasing execution time, but the precise value of slack is unknown.

The implicit-slack predictor works by dividing instructions into *slack bins*, according to the resources that these instructions can tolerate. The number of bins is determined by the number of decisions a control policy must make for each instruction. For an example, let us consider a machine that has two pipelines, one fast and one slow. Let's say the control policy for this machine must make two decisions for each instruction i: **(1)** should i be steered to the fast or slow pipeline? and **(2)** should i be scheduled with high priority or low priority within a pipeline? These two decisions lead to four slack bins:

1. steer to fast pipeline & schedule with high priority,
2. steer to fast pipeline & schedule with low priority,
3. steer to slow pipeline & schedule with high priority,
4. steer to slow pipeline & schedule with low priority.

These four bins can be viewed as corresponding to four *virtual* heterogeneous resources, where each dynamic instruction is assigned to one resource. In general, if a control policy must make k decisions for each instruction (with two choices for each decision), we have 2^k virtual resources,

each corresponding to a slack bin.

Notice that, unlike the explicit-slack predictor, measuring slack implicitly avoids the need for dedicated logic to artificially delay an instruction. Instead, the slack analyzer can delay the instruction *naturally*, by steering it to the resource whose latency needs to be tolerated. An implication of this characteristic is that the precise amount of slack required of an instruction to belong to each bin does not need to be explicitly known: the actual delays experienced by the instruction in the hardware are used to control policy decisions. Also, training is much faster, since there are a relatively small number of bins.

Cost Analyzer. It is theoretically possible to determine the cost of a micro-operation in hardware in a way similar to the "delay and observe" approach employed by the slack analyzer above. The idea would be to use the property exploited in the cost calculation algorithm of Section 5.3. Specifically the cost of an edge e is the minimum of the slacks of all edges parallel to e. Thus, using the explicit slack analyzer above along with a mechanism to determine which events are parallel to others would be enough to determine the cost of an event. Then, computing interaction costs would require determining the cost of groups of events.

Unfortunately, computing cost in this way places even more demands on locality than did the explicit slack predictor, and we have already discussed above why computing slack may be intractable due to the variance of microarchitectural behavior. In the next section, we discuss a method to enhance performance counters to such a degree that a microexecution graph can be constructed offline. With this graph, cost (as well as any other desired metric) becomes possible to compute accurately.

6.3 Shotgun Profiling

While the above analyzers for criticality and slack are needed for quick-feedback online optimization, they are much less flexible than the graph analysis we can do in software. For example, the apportioning policies used in the slack analyzer would need to be very simple, and we have not discussed any analyzers that can measure cost and interaction cost.

To enable these and other advanced measurements of programs executing on real hardware, we propose an improved version of traditional hardware counters. Instead of simply recording the events that occurred to a specific instruction (*e.g.*, as in the ProfileMe infrastructure), we would also record a few bits of *context* around the instruction. This context could include information about whether branch mispredicts or cache misses occurred in the instruction's vicinity.

Offline, we can then build dependence graphs for statistically representative fragments of the execution using the collected information. The context helps in the process by identifying which instruction samples should be placed in the same graph fragment. We call this procedure **shotgun profiling** due to its similarity to shotgun genome sequencing [40]. Once the graph is constructed, we can perform any measurement that is desired, using the software algorithms described in Chapter 4.

The task of enhancing performance counters with enough information such that the graph can be constructed while introducing relatively little hardware complexity is very challenging. Rather than immediately presenting our final solution, we will describe an evolution of designs that will provide a better understanding of the motivation behind our approach, discussed in full at the end.

Design	Problem	Solution
1. Hardware-intensive measurement.	Hardware too expensive since it generates information too rapidly.	*Sample* instructions sparsely.
2. Sample each static instruction once.	Doesn't distinguish between different microarchitectural behavior (*e.g.*, an iteration of a loop with a branch misprediction versus an iteration without one.)	Use microarchitectural context (in the form of a signature).
3. Record short microarchitectural signature around each sampled instruction.	Accumulates error as each instruction is stitched together to form a graph.	Use *long* signature that spans length of graph.
4. Record long signature as a baseline and patch in sample profiles using short signatures.		

Table 6.3: **Profiler designs.** Design #4, with both long and short signatures, is our final, recommended design.

6.3.1 Design #1: The Hardware-Intensive Approach

The conceptually simplest design would be to collect detailed latency and dependence information for every dynamic instruction as it flows through the machine, as is done in a simulator. The detailed information would be enough to construct all of the nodes and edges for each dynamic instruction, such that software could easily construct the graph offline. The exact information required will depend heavily on the processor implementation. For the simulated processor used in this paper, the information in Table 6.4 is sufficient.

Although this approach would be as accurate as constructing the graph in the simulator, it is not reasonably implementable. The primary reason is that the *density* of information collection is too great, in that too much data needs to be collected simultaneously. To measure just one latency for every instruction would require a counter for each instruction in the machine at any one time — and to collect all of the information in Table 6.4, many such latencies would need to be measured. Furthermore, moving all of this information through the machine would require many wires, which could easily cause serious routing problems.

dependence	col	latencies	col
In-order dispatch (*DD*)	S	icache misses, itlb misses	D
Finite fetch bandwidth (*FBW*)	S	constant latency (1 cycle)	S
Finite re-order buffer (*CD*)	S	constant latency (0 cycle)	S
Control dependence (*PD*)	D	branch recovery latency	S
Execution follows dispatch (*DR*)	S	constant pipeline latency	S
Data dependences (*PR*)	reg: S, mem: D	constant latency (0 cycle)	S
Execute after ready (*RE*)	S	functional unit contention	D
Complete after execute (*EP*)	S	Execution latency	D
Cache-line sharing (*PP*)	D	constant latency (0 cycle)	S
Commit follows completion (*PC*)	S	constant pipeline latency	S
In-order commit (*CC*)	S	store BW contention	D
Finite commit bandwidth (*CBW*)	S	constant latency (1 cycle)	S

Table 6.4: **How dependences and latencies are collected when constructing the graph.** 'D' stands for dynamically, 'S' for statically. Dependences and latencies that must be determined dynamically are measured in hardware. Those that can be determined statically are inferred from the program binary (*e.g.*, register data dependences) or the machine description (*e.g.*, fetch and issue bandwidths). Besides the information above, a detailed sample also contains the PC of the instruction and the target address of indirect branches.

From this observation, we derive the most important constraint on the hardware: instructions should be profiled *sparsely*, in a sampling manner. This significantly reduces the amount of hardware required, since there would need to be counters for only a single instruction currently in the machine, as opposed to all of them. The wire count is thus dramatically reduced, with corresponding less affect on the hardware design. This design decision was also made for many current performance counter designs as well as the most popular proposals for enhancements, *e.g.*, ProfileMe [28].

The sparse sampling constraint does make the task much more challenging, however. Remember our goal was to construct graph fragments of the actual execution, which include the nodes and edges representative of a sequence of dynamic instructions. How can we obtain a representation of a sequence of dynamic instructions if we can only sample one instruction in that sequence?

The key, of course, is to exploit the temporally locality present in software (that the same dynamic sequences of instructions are executed over and over again). As we will show in the next design, however, we can not exploit this characteristic in the typical manner of simply mapping measurements of static instructions back to their dynamic equivalents. Instead, a new way of thinking about performance counters is required.

6.3.2 Design #2: One sample per static instruction

In the second design, the hardware measures only one instruction at a time and software periodically retrieves the collected information. The software maintains a data structure indexed by the PC of the instruction, recording each sample in its appropriate entry. Then, the software selects a sequence of dynamic instructions from the binary (in a random sampling manner) and constructs a graph fragment using the information collected by the hardware and stored by PC. The assumption is that different dynamic instances of the same static instruction will exhibit similar microarchitectural behavior, so that the graph will accurately represent actual program execution.

Unfortunately, we found that graph fragments constructed in this way are not representative: empirically, the icosts computed are typically off by a factor of two or more when compared to those computed in the simulator. The problem is that the assumption that different dynamic instances of an instruction exhibit similar behavior is not a good one. As an example, consider Figure 6.8. In the first iteration of the loop, the instruction at PC 0x30 experiences an icache miss, while on the second iteration it does not. Thus, the graph for the first iteration is different than the graph for the second iteration, even though the same static code is executed (specifically, the DD edge latency is different).

The obvious lesson here is that variations in the microexecution need to be distinguished

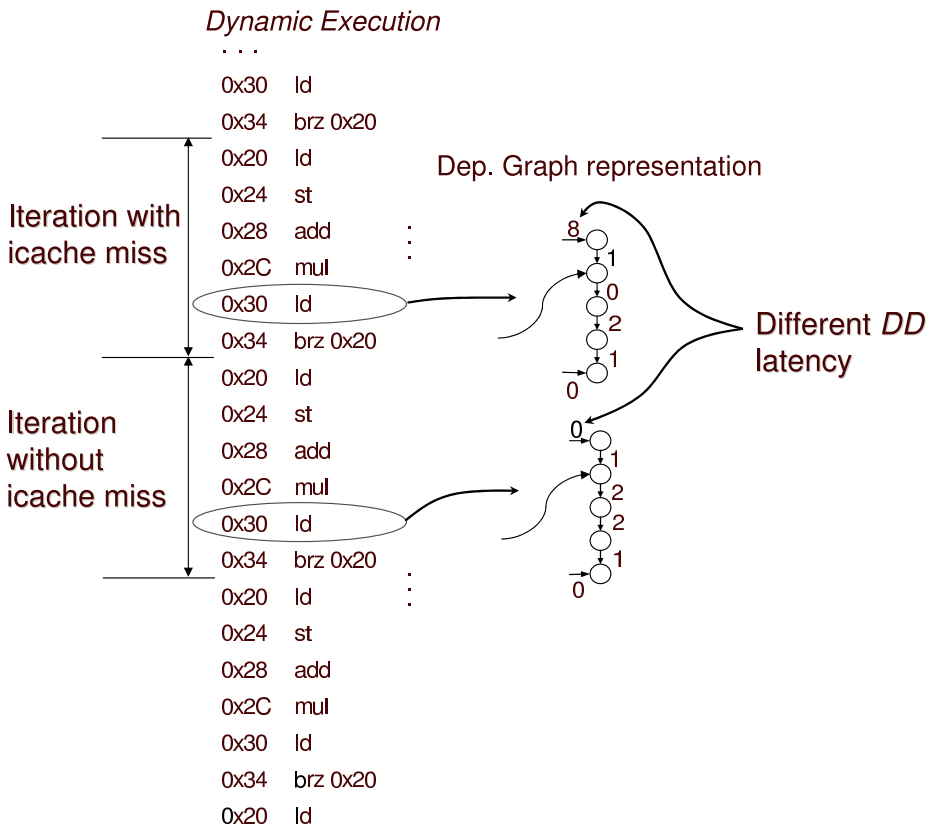


Figure 6.8: **Same static code, different microexecutions.**

in order to construct accurate graph fragments. In other words, multiple profiles for each static instruction need to be maintained. Specifically, we should ideally maintain one profile for each *microarchitectural context*, *e.g.*, in the example above, one sample for each instruction in both (a) an iteration with the icache miss and (b) an iteration without.

For the next two design proposals, the primary goal will be to develop an *inexpensive* way to distinguish between microarchitectural contexts.

6.3.3 Design #3: Shotgun profiler, only short signatures

We distinguish between microarchitectural contexts by adding a *signature* to each sample collected from the hardware. The signature distinguishes between contexts by encoding microarchitectural events and state that surrounds the single dynamic "target" instruction. Thus, each sample consists of two things: **(i)** detailed latency and dependence information about the target instruction and **(ii)** a signature surrounding that instruction. If the signatures of two samples match, we assume the samples are from the same context.

The signature should uniquely identify the microexecution context while keeping the hardware cost as low as possible. More specifically, whenever the signatures for two samples are the same, the detailed latency and dependence information for the target instruction should also be the same. For our design, we chose to record two bits per dynamic instruction for ten instructions before and after the targeted instruction. The two bits are an experimentally determined hash of microarchitectural context, specified in Table 6.5. A more detailed description of the process for designing these bits is below.

The graph construction algorithm uses the signatures to determine which samples should be placed side-by-side within a graph fragment. As an example, consider Figure 6.9a. Two samples

Bit	When to set to '1'
1	Set to 1 if the instruction is a (1) *taken* branch or (2) load or store. Reset to 0 if L2 dcache miss.
2	Set to 1 if the instruction experiences a (1) L1 or L2 icache miss, (2) L1 or L2 dcache miss, (3) tlb miss, or (4) branch mispredict.

Table 6.5: **Description of signature bits.** The signature bits are meant to distinguish between different microarchitectural contexts. Experimentally, we determined the above hash function produced good results. Intuitively, the hash works well because it distinguishes between the most important events that occur in the microprocessor. For a different processor implementation than the one assumed in our simulator, a different signature might be required, perhaps one that uses more than two bits per dynamic instruction.

are taken; in this case, they are of two different static instructions from two iterations of the same loop. By finding overlap among the appropriate signature bits between the two samples, we see that they "fit" together. Thus, they come from iterations of the loop with the same context and should be placed together in the graph fragment. By repeatedly applying this matching process, we can construct a graph fragment of arbitrary size.

This algorithm is very similar to a popular algorithm for DNA sequencing, called *shotgun sequencing* [40] (see Figure 6.9b). Due to the similarity, we refer to the general class of profilers which use signatures as *shotgun profilers*. There is a large space of possible algorithms and infrastructures that exploit shotgun profiling, only a couple of which are presented in this paper.

Returning to the example of Figure 6.8, consider how a signature could help distinguish between loop iterations with different behavior. For the first iteration of the loop, an icache miss will appear in the signature; while in the second iteration it will not. Thus, the samples with the icache miss will be attached together in one portion of the graph fragment while the samples without the miss will be in another portion.

Empirically, we have found this design reduces the error by two to four times over one that does not distinguish between different microexecution behavior. Nonetheless, the performance

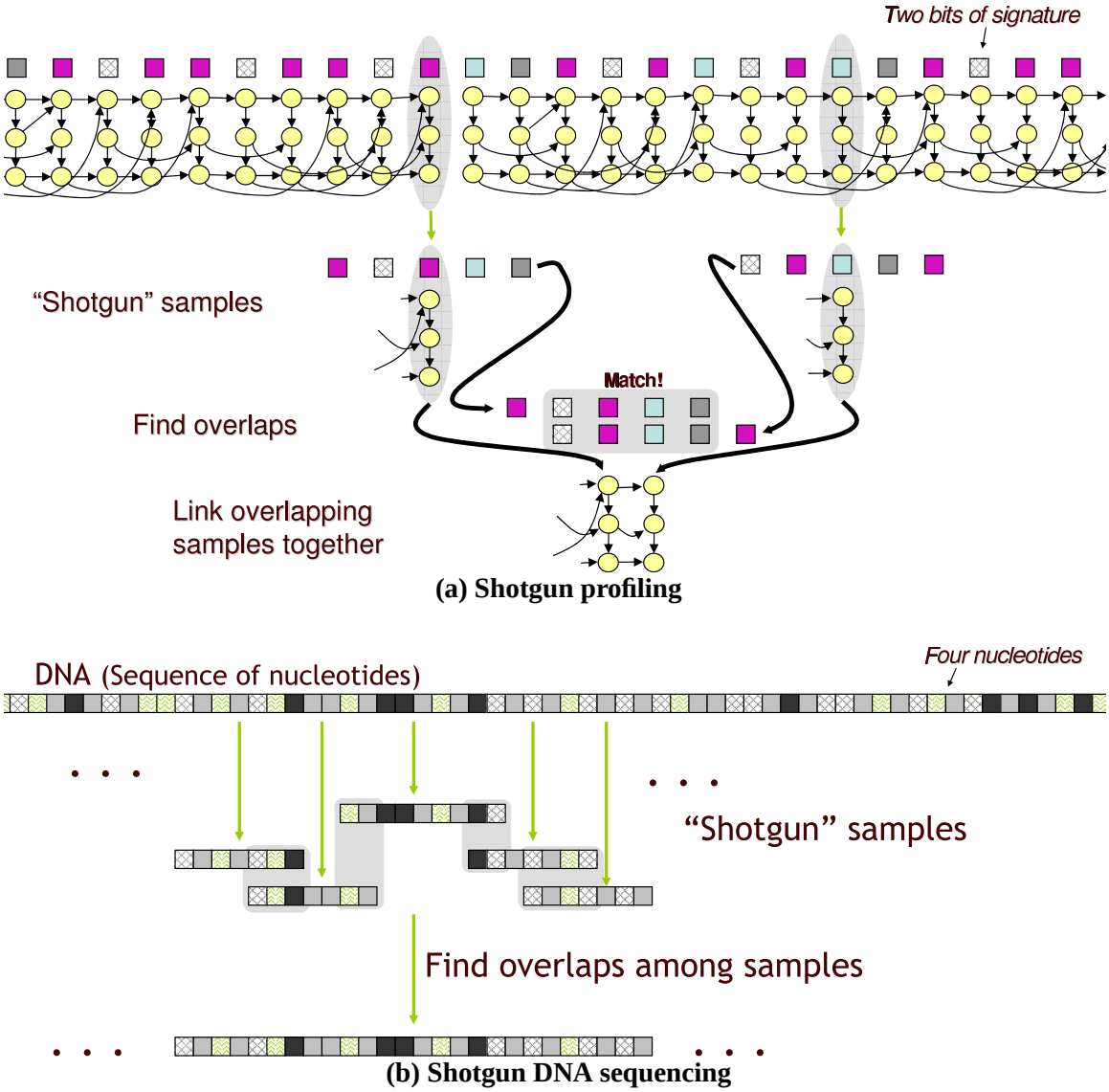


Figure 6.9: **Shotgun profiling and DNA sequencing (a)** The shotgun profiler works by collecting random "shotgun" samples that include a signature and detailed information about a single instruction. These samples are placed in a database and, offline, graph fragments are constructed by finding overlaps among the signatures of different samples. Our design uses a signature with two bits for each of the ten dynamic instructions before and after the target instruction. For illustration, the figure uses a smaller signature. **(b)** DNA researchers face a problem similar to ours. Instead of constructing a graph, they seek to determine the sequence of nucleotides that comprise a strand of DNA. Their measurement apparatus, however, cannot simply observe the entire sequence at one time. Instead, they can only observe short, random, samples of the overall sequence. Their solution to this problem is called "shotgun" sequencing. First, many random samples are collected using their measurement apparatus. Then, offline, the full DNA sequence is constructed by looking for *overlaps* among the small fragments.

is still far from acceptable. The reason is that error accumulates for each sample placed into the graph, for a couple of reasons:

- *Missed correlation of distant events.* The context is only of nearby instructions, over a range of twenty instructions. If, for instance, the latency of an instruction is affected by an event that occurs forty instructions away, this correlation cannot be captured. Since modern machines exploit parallelism across a rather large range of instructions, this effect can be significant.

- *Missing samples.* If an exact signature match cannot be found, the closest approximate match is used. In our experience, the missing samples are the ones with the rarest signatures, since they have the lowest probability to be collected. This causes rare events (*e.g.*, branch mispredictions) to be under-represented in the constructed graphs. Collecting more samples would reduce the error, but considering the exponential number of possible signatures, it may be infeasible to collect sufficiently many to eliminate the error.

To improve over this design, we need to reduce the accumulation of error. In the next section, we do this by adding a stable microarchitectural context "skeleton" on top of which the graph is constructed.

6.3.4 Our final solution: Shotgun profiler, long and short signatures

Our final and recommended design introduces a second type of sample to be collected by the hardware, in addition to the one collected in design #3. The new sample is called a *signature sample* and consists of a single "start" PC and the two signature bits for each of the next 2000 dynamic instructions. Signature samples are a natural way to identify correlation between distant events, and, as we'll show below, can also mitigate the effect of missing samples.

The software graph construction algorithm works by first selecting a long signature sample at random, which serves as a "skeleton" for the graph to be built. (The random selection ensures each signature sample is chosen with equal probability, which naturally gives priority to hot microexecution paths.) The goal of the algorithm is to fill in this skeleton with *detailed samples* to form a latency-labeled dependence graph. A detailed sample is identical to the samples collected in design #3 above. To construct the graph, a detailed sample is selected for each dynamic instruction in the signature sample, where the selection is based both on a PC match and a signature match.

For example, consider building the graph nodes for the first instruction in the signature sample of Figure 6.10. The first instruction has PC of $0x24$, so we look up detailed samples with this PC. Then, we select the one whose signature bits match the corresponding bits in the signature sample. Finally, the nodes for this instruction are constructed from the selected detailed sample.

If no detailed samples for the PC are found at all, which empirically happens less than 2% of the time, we infer what we can from the signature sample and the binary, using default values for unknown latencies. For example, if bit two of the signature is set to one and we know from the binary the instruction is a branch, we will infer that the branch was mispredicted. (In this instance, it is possible that an icache miss occurred instead of the branch mispredict, but we would guess a branch mispredict occurred for branch instructions.) Here, we see one advantage of the signature sample design over design #3: the signature sample gives us some information (*e.g.,* whether a branch mispredict occurred) even when no matching detailed sample has been collected.

If some detailed samples are found, but none have an exact signature match, the detailed sample with the closest match is selected. An inexact match may reduce accuracy for that selection, but (unlike design #3) the signature sample provides a stable skeleton for future matches. Thus,

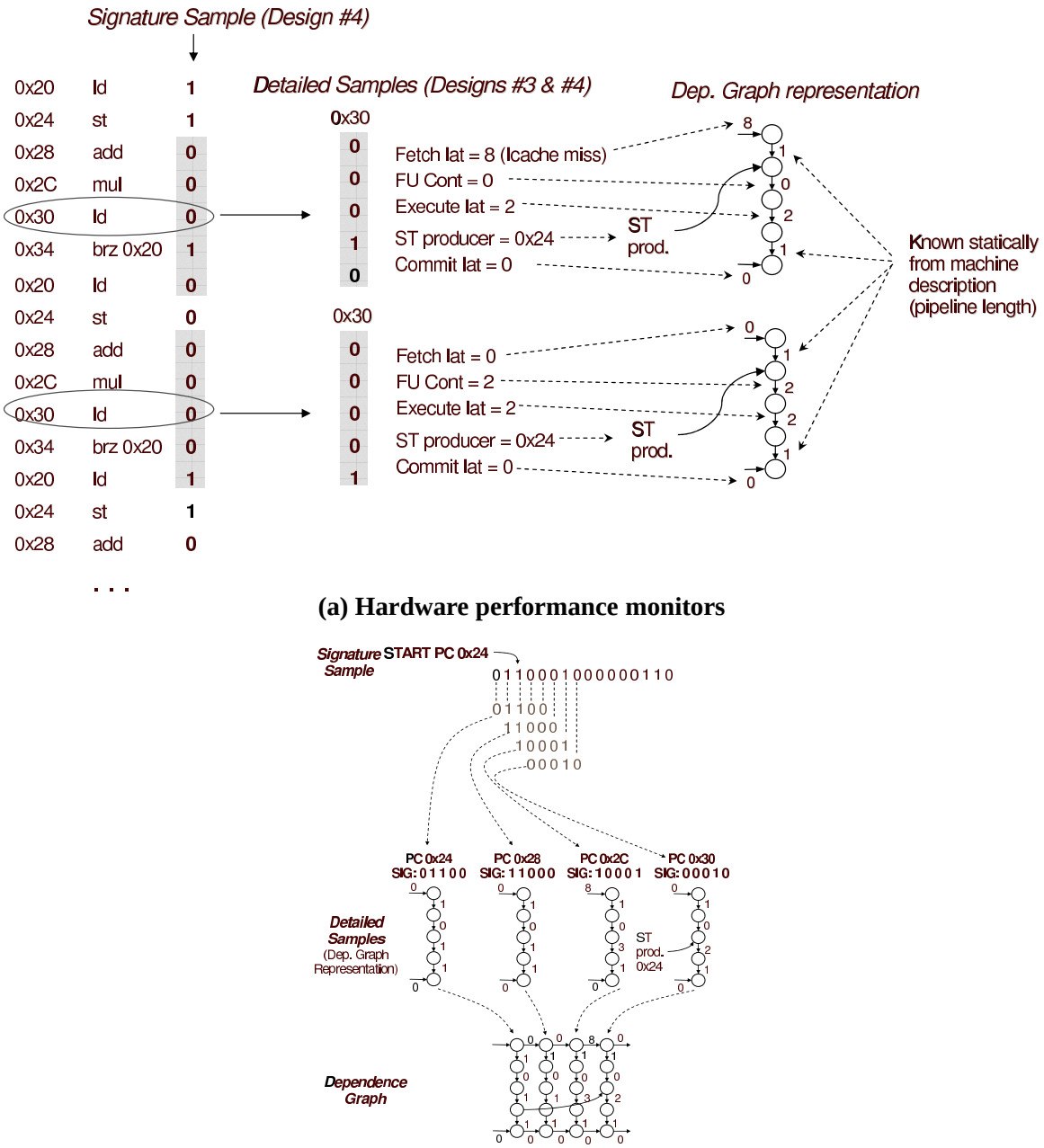


Figure 6.10: **The profiler infrastructure consists of two parts. (a) Hardware performance monitors.** Our hardware performance monitors collect two types of samples: signature samples and detailed samples. For illustration, the figure shows one signature bit per instruction and collection of the bits for two instructions before and after each detailed sample. For greater accuracy, our design uses two signature bits per instruction (see Table 6.5) and collects signature bits for ten instructions before and after each detailed sample (see Figure 6.11a). **(b) Post-mortem software graph construction.** The dependence graph is constructed by concatenating detailed samples, so that the resulting graph is representative of the microexecution denoted by the signature sample.

a single mismatch doesn't cause error to propagate through the rest of the graph. The complete algorithm for constructing a graph fragment is in Figure 6.11.

1. Randomly select a *signature sample* for the skeleton.
   Call the starting PC in this sample the $StartPC$.
2. For each instruction i from $StartPC$ to end of fragment
2a. Get from database all detailed samples with i's PC.
2b. Select the detailed sample whose signature bits most closely matches the portion of the signature sample 10 instruction before i to 10 instructions after. The closeness of a match is judged by the number of identical bits.
2c. Append sample's nodes and edges to the graph (see Fig. 6.10).
2d. Determine PC of next instruction, $i+1$ (call PC of i $CurPC$ and PC of $i+1$ $NextPC$):
2d1. If i is not a branch, $NextPC \leftarrow CurPC + 4$
2d2. If i is a direct branch and signature bit 1 of i is 1,
Compute branch target and set $NextPC$ equal to it
Else $NextPC \leftarrow CurPC + 4$
2d3. If i is a call, push target PC onto stack
For returns, pop stack (if nonempty) and set $NextPC$ to that PC
2d4. If i is an indirect branch, set $NextPC$ equal to target PC in detailed sample for i
2e. Check for illegal signature bit/opcode combinations (see text).

Figure 6.11: **Algorithm for constructing a graph fragment in software.**

Determining PCs

Remember that a signature sample consists solely of a start PC and the signature bits, *i.e.*, to reduce hardware costs the PCs of other instructions are not recorded. Thus, we need to use some intelligence to infer the PC of each dynamic instruction in the signature sample. For direct conditional branches, we include the branch direction in the signature bits and lookup the binary for the target address of taken branches.

For indirect branches, we include the branch target address in the detailed samples. Assuming a signature match is a good indication of which target address an indirect branch will resolve to, the normal matching procedure described above will yield the correct next PC. We have found, empirically, that this procedure yields the correct target address most of the time, for 60–99% of the

indirect branches, depending on the benchmark. (Note that this accuracy is highly dependent on the choice of signature; other signatures, perhaps using more bits, could achieve greater accuracy.)

In the cases where the matching sample's target address is not correct, there could be serious error in the graph fragment construction. To mitigate the error, we take advantage of the fact that some combinations of opcodes and signature bits could never occur down a correctly determined path. For instance, if an instruction on the long signature sample has its first bit set to one, it should be a load, store, or branch. If the computed PC (step $2d$ in the algorithm) does not correspond to one of these instruction types in the program binary, we know there is an inconsistency and abort building the graph segment — building such a graph would lead to error in the results. We have found that 95–100% of errant graphs are indeed discarded using this technique.

Finally, note that for return instructions whose call counterpart occurs within the graph fragment, a stack of call addresses can provide the correct target address. If the call counterpart is outside the graph fragment, a return is treated the same as an indirect branch.

6.3.5 Measuring profiler accuracy

In this section, we measure the accuracy of the shotgun profiler. For the baseline, we use the *multiple-simulation approach*, which computes the cost of a set of events S by comparing the execution time reported by a normal simulation to that of a simulation with all the events in S idealized. For example, for the category labeled "bmisp+dmiss", a simulation is run where (simultaneously) all branch mispredictions are made correct and all loads hit in the level-one cache. The result from the multiple-simulation approach is then compared to that obtained through analysis on the dependence graph constructed by the profiler.

We find that the profiler's error in icost measurement is, on average, 9% off of the baseline,

as measured via multiple simulations. From the breakdown of error sources, we found that the modeling of the microprocessor as a dependence graph contributed more error than either the sparse sampling or the profiler algorithm. A more thorough discussion follows below.

Discussion of category errors

Tables 6.6 and 6.7 shows breakdowns computed with the profiler relative to multiple simulations for the categories in Table 7.5(a). A couple of observations can be made from the breakdowns. First, the type of interaction (parallel or serial) is always the same with the profiler as the *multisim* baseline. Second, the profiler comes very close to the *multisim* baseline most of the time, typically with error less than a few percent of the overall execution time.

There are some examples, however, where the error in the icost calculation is substantial. One category that tends to exhibit significant error for some benchmarks is the instruction window (*win*). For example, for gap, the error is -11.3% and for $vortex$, it is -8.4%. The cause of this error is the profiler's inability to completely accurately idealize the instruction window. Specifically, since the graph fragments constructed by the profiler are of finite size, it is not possible to accurately model a very large instruction window — needed when performing the idealization. Thus, the effective window size modeled by the profiler for idealization purposes will be smaller than that of the simulator, and thus it will likely under-predict the window's cost. This error could be reduced by increasing the size of the graph fragments constructed.

Sources of error

In Table 6.8 we attempt to understand the sources of error in the profiler. To this end, the breakdowns of Table 7.5(a) are computed in four different ways. *multisim* is the baseline, as above.

	bzip			crafty			eon		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	20.3	23.2	+2.9	23.4	24.2	+0.8	17.0	17.7	+0.7
win	15.9	15.5	-0.4	17.3	15.4	-1.9	18.2	15.2	-3.0
bw	**6.5**	**3.9**	**-2.5**	**8.7**	**6.7**	**-2.0**	**10.5**	**6.6**	**-3.9**
bmisp	37.3	38.3	+1.1	26.0	24.1	-1.9	14.2	14.4	+0.2
dmiss	23.3	23.5	+0.2	6.9	6.5	-0.4	0.8	0.6	-0.2
shalu	8.9	10.0	+1.1	10.7	11.2	+0.5	4.5	5.2	+0.7
lgalu	0.3	0.3	+0.0	0.7	0.8	+0.1	12.6	12.1	-0.5
imiss	0.0	0.2	+0.2	0.7	0.2	-0.5	9.2	8.7	-0.5
dl1+win	-4.8	-5.2	-0.5	-11.5	-11.7	-0.2	-7.7	-7.2	+0.5
dl1+bw	6.9	5.9	-1.2	10.0	10.5	+0.5	6.9	6.8	-0.1
dl1+bmisp	-9.1	-9.6	-0.4	-4.9	-4.2	+0.7	-3.8	-3.9	-0.1
dl1+dmiss	-0.8	-0.7	+0.1	-0.4	-1.3	-0.9	-0.2	-0.3	-0.1
dl1+shalu	-3.5	-4.3	-0.8	-4.0	-4.5	-0.5	-0.6	-1.0	-0.4
dl1+lgalu	-0.2	-0.3	-0.1	0.3	0.2	-0.1	-0.5	-0.8	-0.3
dl1+imiss	0.0	0.0	+0.0	0.0	0.0	-0.0	1.3	1.0	-0.3

	gap			gcc			gzip		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	12.6	12.6	+0.0	17.4	17.0	-0.4	29.9	31.7	+1.8
win	**41.2**	**29.9**	**-11.3**	14.4	13.0	-1.4	14.7	13.1	-1.6
bw	4.1	2.4	-1.7	**9.0**	**7.1**	**-1.9**	6.6	5.5	-1.1
bmisp	11.3	11.4	+0.1	23.9	21.5	-2.4	23.8	23.4	-0.4
dmiss	22.6	21.8	-0.8	25.5	27.7	+2.2	8.1	7.8	-0.3
shalu	13.8	11.2	-2.6	5.4	4.7	-0.7	18.9	20.7	+1.8
lgalu	5.3	5.7	+0.4	0.6	0.2	-0.4	0.5	0.5	+0.0
imiss	1.3	0.9	-0.4	2.1	1.4	-0.7	0.1	0.0	-0.1
dl1+win	-6.3	-6.1	+0.2	-4.1	-3.5	+0.6	-9.3	-9.6	-0.3
dl1+bw	3.0	3.3	+0.3	10.9	12.4	+1.5	6.2	5.7	-0.5
dl1+bmisp	-2.9	-2.7	+0.2	-6.3	-5.4	+0.9	-3.6	-3.1	+0.5
dl1+dmiss	0.4	0.3	-0.1	-0.9	-1.4	-0.5	-0.2	-1.3	-1.1
dl1+shalu	-0.3	-2.1	-1.8	-2.1	-1.4	+0.7	**-7.6**	**-9.4**	**-1.8**
dl1+lgalu	-0.2	-0.5	-0.3	-0.5	-0.2	+0.3	-0.5	-0.5	-0.0
dl1+imiss	0.3	0.4	+0.1	0.3	0.2	-0.1	-0.0	-0.0	+0.0

Table 6.6: **Measuring accuracy of profiler.** Continued in Table 6.7.

	mcf			parser			perl		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	**7.1**	**7.4**	**+0.3**	17.9	19.1	+1.2	30.7	31.3	+0.6
win	4.8	4.3	-0.5	**17.1**	**13.2**	**-3.9**	6.2	5.6	-0.6
bw	0.6	0.4	-0.2	4.0	3.0	-1.0	10.3	8.1	-2.2
bmisp	25.3	25.1	-0.2	15.8	14.9	-0.9	35.4	38.0	+2.6
dmiss	80.8	79.0	-1.8	32.1	28.1	-4.0	1.3	0.8	-0.6
shalu	1.4	1.4	+0.0	17.9	17.1	-0.8	7.4	8.2	+0.8
lgalu	0.0	0.0	+0.0	0.1	0.1	-0.0	0.7	0.6	-0.1
imiss	-0.0	-0.0	+0.0	0.1	0.1	+0.0	**5.3**	**2.7**	**-2.6**
dl1+win	-0.0	-0.1	-0.1	-6.3	-6.2	+0.1	-5.9	-5.4	+0.5
dl1+bw	0.4	0.3	-0.1	4.9	4.9	-0.0	9.9	9.7	-0.2
dl1+bmisp	-2.3	-2.3	-0.0	-2.5	-2.4	+0.1	-8.4	-8.2	+0.2
dl1+dmiss	-0.4	-0.5	-0.1	-0.9	-1.7	-0.8	-0.1	-0.1	-0.0
dl1+shalu	-0.2	-0.1	+0.1	-4.1	-4.9	-0.8	-2.2	-2.0	+0.2
dl1+lgalu	0.0	0.0	-0.0	-0.1	-0.0	+0.1	-0.7	-0.5	+0.2
dl1+imiss	0.0	0.0	+0.0	-0.0	-0.0	+0.0	1.0	0.6	-0.4

	twolf			vortex			vpr		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	**17.1**	**19.2**	**+2.1**	27.4	30.4	+3.0	18.5	20.3	+1.8
win	24.2	22.3	-1.9	42.8	34.4	-8.4	22.9	21.9	-1.0
bw	4.5	3.5	-1.0	**8.0**	**5.3**	**-2.7**	**5.9**	**4.4**	**-1.5**
bmisp	22.2	22.6	+0.4	1.5	0.8	-0.7	23.4	23.1	-0.3
dmiss	34.3	34.3	-0.0	19.8	18.7	-1.1	32.5	32.1	-0.4
shalu	7.7	7.7	-0.0	3.9	5.4	+1.5	7.3	8.2	+0.9
lgalu	4.2	4.2	+0.0	1.5	1.5	-0.0	4.1	4.0	-0.1
imiss	0.1	0.0	-0.1	3.3	0.9	-2.4	0.0	0.0	-0.0
dl1+win	-3.6	-4.5	-0.9	-25.7	-27.0	-1.3	-6.2	-6.9	-0.7
dl1+bw	1.7	1.5	-0.2	17.7	17.7	+0.0	1.9	2.1	+0.2
dl1+bmisp	-5.8	-5.8	+0.0	-0.2	-0.1	+0.1	-4.6	-4.4	+0.2
dl1+dmiss	-0.1	-1.9	-1.8	-1.6	-1.2	+0.4	-1.4	-2.2	-0.8
dl1+shalu	-0.5	-0.3	+0.2	-3.3	-4.7	-1.4	-1.5	-1.9	-0.4
dl1+lgalu	-0.0	-0.1	-0.1	-1.2	-1.3	-0.1	-0.3	-0.6	-0.3
dl1+imiss	-0.0	-0.0	+0.0	0.5	0.1	-0.4	0.0	0.0	+0.0

Table 6.7: **Measuring accuracy of profiler.** (Continued from Table 6.6.) Validation was performed on the same CPI contribution breakdown (with results expressed in percent of total CPI) as in Table 7.5(a). The *multisim* column shows the value for each category computed through the multiple simulation approach. This serves as the baseline for measuring accuracy. The *profiler* column shows the values the profiler computed, while the *error* column is the difference between the *profiler* and *multisim*. The single largest percent error (considering categories greater than 5%) for each benchmark is in bold.

	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
multisim→fullgraph	11.1	7.0	9.1	8.4	8.6	14.3	2.2	4.9	7.9	5.1	9.7	9.0
fullgraph→graphfrag	3.6	2.8	3.5	3.2	3.1	2.1	0.2	3.3	2.9	2.4	4.0	2.4
graphfrag→profiler	4.9	3.4	2.3	3.7	10.6	3.9	0.1	2.1	5.4	3.4	4.6	5.0
multisim→fullgraph	11.1	7.0	9.1	8.4	8.6	14.3	2.2	4.9	7.9	5.1	9.7	9.0
multisim→graphfrag	12.9	7.8	11.0	8.9	9.5	13.9	2.4	6.9	9.8	6.0	13.0	9.4
multisim→profiler	**11.1**	**7.8**	**9.5**	**8.9**	**11.7**	**9.3**	**2.5**	**9.0**	**12.6**	**3.7**	**12.4**	**9.2**

Table 6.8: **Sources of errors for the shotgun profiler.** The breakdowns of Table 7.5(a) were computed four ways to better understand the sources of error in the profiler. *multisim* is the breakdown computed via multiple simulations; it serves as the baseline for comparison. *fullgraph* indicates the dependence graph of the entire program was used, as in Section 7.3.1; *graphfrag* is the breakdown computed assuming the graph fragments constructed by the profiler were perfect; and *profiler* is the breakdown as computed on the imperfect graph fragments actually constructed by the profiler (described in Section 6.3). The numbers presented are the average percent difference in the categories (excluding categories under 5%) between the two schemes in the first column of each row. For instance, the *multisim→fullgraph* row is determined by computing $abs(\textit{multisim} - \textit{fullgraph})/(\textit{multisim})$ for each category over 5% and averaging the results. Note that the *multisim→profiler* row is the total error for the profiler.

fullgraph is the breakdown computed with the dependence graph of the entire program, just as was done for the results of Section 7.3.1. *graphfrag* is the breakdown computed assuming the graph fragments constructed by the profiler were perfect (*i.e.*, exactly as they exist in the full graph), and *profiler* is the breakdown as computed on the imperfect graph fragments actually constructed by the profiler (using the signature-based algorithm).

The first series of measurements examines the accuracy of each step of the full profiling scheme. *multisim→fullgraph* is the error introduced by modeling the machine as a dependence graph, as opposed to using a detailed simulator. Typically, this error is less than 10%; but, nonetheless, it does often contribute the largest fraction of the overall error of the profiler. It can potentially be reduced by increasing the detail of the model to include currently unmodeled aspects of the microarchitecture, such as contention for memory busses.

The *fullgraph→graphfrag* row shows the error caused by measuring the breakdowns using only a relatively small number of graph fragments as opposed to the entire graph. This *sampling*

error is a significant component of the overall error for some benchmarks, *e.g.*, $vortex$. The good news here is that this error can be reduced by simply running the program longer to collect more samples.

The *graphfrag→profiler* row shows the error introduced by the profiler's signature-based algorithm for constructing graph fragments. The error is due to two factors: (1) the signature not being sufficient to identify the correct detailed sample to paste into the graph and (2) a signature-matching detailed sample not being in the database. The second error factor can be reduced by simply collecting more samples, while the first requires some redesign of the signature bits.

For most benchmarks, the signature-based algorithm contributes only a modest amount to the error, typically less than 5%. An exception is gcc, with an error of 10.6%. Upon closer inspection, we found that this large error is primarily due to the target address of indirect branches not being determined correctly, leading to many graphs being discarded (see Section 6.3.4). One way to reduce the error would be to construct smaller graph fragments, so that the probability of encountering a difficult indirect branch in any one fragment is reduced. We found that reducing the fragment size from 2000 to 1000 reduced the error to 5.1% (but, averaged over all benchmarks, the larger size improved accuracy). Another method would be to enhance the signature to improve its ability to distinguish indirect branch targets, *e.g.*, by adding an additional bit that is set equal to one of the bits of the PC.

The second series of measurements shows the error of three of the breakdown computations — *fullgraph*, *graphfrag*, and *profiler* — relative to *multisim*. The purpose of these measurements is to show how each individual source of error contributes to the overall error of the profiler. Notice that the overall error is not always monotonically increasing as each new source of error is in-

cluded. For example, the *multisim→graphfrag* error for eon is 11.0%, while the *multisim→profiler* error is less, 9.5%. The reason is that the error introduced at each stage could be positive or negative, independent of the direction of errors at previous stages. Thus, it is statistically likely that the errors will compensate sometimes. In the case of the example, for eon, the *graphfrag→profiler* error was mostly in the opposite direction of the errors in the previous two stages.

The overall error for the profiler is shown in the last row of Table 6.8, labeled *multisim→profiler*. The range of errors for the benchmarks is from 3% (for mcf) to 13% (for $perl$), with the average error being 9%. Since the ability to compute costs and icosts from hardware profiles is qualitatively new, standards for accuracy have not been set; but an error of 9% seems small enough to perform meaningful analysis. If a smaller error is desired, increasing the precision of the graph model appears to offer the greatest opportunity for improvement.

6.4 Summary

In this section, we have discussed hardware mechanisms for (1) detecting criticality and slack, (2) predicting the criticality and slack of future instructions based on past detections, and (3) a profiler designed to overcome the limitations of performance counters by providing insights into how parallelism affects program performance. The slack and criticality predictors are designed for quick "turnaround": the characteristics are quickly detected and recorded for use later in the same program run. The profiler, however, collects information during a program run for offline graph analysis. It may be possible to use the result of the analysis in the same program run that the information was collected (*e.g.,* via a dynamic optimization system), but, in any case, the turnaround time is much longer. The gain from the profiler is the much more powerful analysis that it provides

over the purely hardware predictors (*e.g.*, computation of interaction costs). Thus, as we will see in the next chapter, the types of applications that can make use of the slack and criticality predictors have a very different nature than for the profiler.

Chapter 7

Applications of Criticality

In this chapter we discuss how the criticality metrics, hardware, and software algorithms can be put to practical use — improving performance, saving energy, and reducing the amount of human effort required in both hardware and software design. To organize the discussion, we divide the application space into three broad categories:

- *Hardware Control Policies.* Criticality can be very useful in dynamic optimizations by providing intelligent policies for resource arbitration, speculation control, and combating energy and wire delay constraints.

- *Hardware Design.* The ability to produce complete breakdowns of performance, including the contribution to performance of each hardware resource and the interactions between them, can reveal tradeoffs designers did not know even know existed.

- *Software Optimization and Design.* Criticality can point to the most expensive portions of code that need optimization — for example, the most expensive loads to prefetch, the most expensive branches to predicate, and the procedures that could benefit the most from dynamic

code modifications. Moreover, the ability to quickly model the effects of software modifications without actually performing them allows software writers to test out the performance of many different organizations (which is especially important when writing multithreaded programs).

Since the goal of this chapter is to illustrate the many practical uses criticality can have in the real world, we intermix our work with that of others (most of which built upon our foundation). It will be made clear where credit resides for each effort.

7.1 Simulation Methodology

The simulator we used is built upon the SimpleScalar tool set [18] with the majority of the timing model rewritten to better reflect possible next generation microarchitectures. The baseline configuration for the experiments is described in Figure 7.1. Alterations to this configuration are made for particular experiments (*e.g.*, clustered machines) and are mentioned along with the results. We used the SPEC2000int suite as optimized Alpha binaries using reference inputs. Since the reference input runs are too long for practical simulation, and some of our simulations/analyses are very demanding, we performed detailed timing simulation for only a 100 million dynamic instruction segment of each binary. To avoid simulating only initialization code, we skipped the first eight billion dynamic instructions. The caches were then warmed up (over 500 million instructions) before beginning the 100 million instruction detailed simulation run.

Dynamically Scheduled Core	128-entry instruction window, 6-way issue, 15-cycle pipeline, perfect memory disambiguation, fetch stops at second taken branch in a cycle.
Branch Prediction	Combined bimodal (8k entry)/gshare (8k entry) predictor with an 8k meta predictor, 4K entry 2-way associative BTB, 64-entry return address stack.
Memory System	32KB 2-way associative L1 instruction and data (2 cycle latency) caches, shared 1 MB 4-way associative 12-cycle latency L2 cache, 100-cycle memory latency, 128-entry DTLB; 64-entry ITLB, 30-cycle TLB miss handling latency.
Functional Units (latency)	6 Integer ALUs (1), 2 Integer MULT (3). 4 Floating ALU (2), 2 Floating MULT/DIV (4/12), 3 LD/ST ports (2).

Table 7.1: **Baseline configuration of simulated processor.**

7.2 Hardware Control Policies

Much of research into computer architecture focuses on new hardware *mechanisms* to improve performance — for example trace caches, grid processors, and larger instruction windows. We, instead, focus on the *policies* that enable these structures to perform well — *e.g.*, replacement policies for trace caches, scheduling policies for grid processors, and policies for deciding which instructions should be allocated slots in the window. Typically, these policies are designed in an ad-hoc manner using heuristics that are tuned over particular benchmarks. Our goal is to develop policies based on criticality analysis that dynamically match the policy to the needs of whatever program is running. We explore three categories of policies:

- *Resource Arbitration.* Criticality can be used to decide which instructions should be allocated scarce resources, *i.e.*, issue slots in an out-of-order processor. Resource arbitration is especially important in the distributed and non-uniform processor architectures being proposed to deal with technological constraints, such as increasing wire delays and energy dissipation.

- *Speculation Control.* Criticality indicates which events could benefit from speculation and which cannot. Since speculation involves overhead and the risk of mis-speculation, intelli-

gently deciding which predictions to make can improve performance. Furthermore, avoiding the risk of misspeculation when little reward is possible reduces the extra energy dissipated by work that must later be squashed.

- *Dynamic Hardware Reconfiguration.* A popular way to reduce energy dissipation, as well as adjust the allocation of hardware resources to meet program demands, is to reconfigure the hardware at runtime.

7.2.1 Resource Arbitration

Resource arbitration is useful whenever there is contention for scarce resources within the processor. This occurs not only when there are fewer resources than desired, *e.g.,* 3 adders when 4 add instructions are ready to execute, but also when resources are available at different "quality levels", *e.g.,* a fast instruction window versus a slow one. Sometimes the resources are at effectively different quality levels even when they are designed identically. For example, in an instruction-level distributed processor, resources close to each other are at a higher quality level than those far apart.

Our primary investigation with using criticality for arbitration was with instruction-level distributed processing (ILDP) [98]. ILDP is important primarily for two reasons: (1) increasing wire delays relative to logic make distributing the architecture a necessity and (2) distributed architectures often have less energy dissipation than their monolithic equivalents. The reasons for the less energy dissipation is that (1) many hardware structures have power requirements that grow quadratically with their size and (2) distributing instruction processing structures enables some portions of the machine to run at different frequencies than other portions.

For our case study, we explore applying criticality to control policies of a modest instance

of ILDP where the instruction window is distributed. Microarchitectures employing simple variations on this design already exist and more advanced versions will likely exist in the near future. In the first set of experiments, we use criticality to steer instructions to the most appropriate *cluster*, where each cluster contains a portion of the machine's instruction window and functional units. The goal in the steering policy is to reduce the effective performance penalty due to inter-cluster communication. We also use criticality for resource arbitration within each cluster, since distributing the functional units will inevitably lead to greater load imbalance and hence greater contention. Together, the criticality-based policies improved performance by up to 21% (10% on average) over the standard register-dependence based policies.

The second set of experiments uses criticality to reduce energy dissipation. The microarchitecture explored has two clusters, one of which is faster (higher quality) than the other. The slower cluster requires less power due to the quadratic relationship between power and clock frequency. Thus the goal is to steer as many instructions to the slow cluster as possible without reducing performance. Slack is a perfect fit for this task since those instructions that have enough slack to afford the slower execution should be steered to the slower cluster. Our slack-based policies reduced the 12–15% performance loss incurred using pre-existing policies to a negligibly small amount.

To illustrate other resource arbitration applications of criticality, we also discuss two related works produced by other researchers. One uses criticality to control a non-uniform cache architecture, where one cache has a higher latency than another. A second study uses slack to more efficiently allocate scarce instruction window slots to instructions as required to maximize performance (as opposed to the normal program-order policy).

Criticality in a Uniform ILDP Architecture

Focused instruction scheduling and *steering* are optimizations that use the critical path to arbitrate access to contended resources (scheduling) and mitigate the effect of long latency inter-cluster communication (steering). The scheduling and steering are focused in the sense that they directly target the computation that needs to be sped up, as opposed to applying the same policies to all computation. Our experiments show that the two optimizations improve the performance of a next-generation clustered processor architecture by up to 21% (10% on average), with focused instruction scheduling providing the bulk of the benefit.

The Problem. The complexity of implementing a large instruction window with a wide issue width has led to proposals of designs where the instruction window and functional units are partitioned, or *clustered* [10, 32, 58, 69, 82]. Clustering has already been used to partition the integer functional units of the Alpha 21264 [46]. Considering the trends of growing issue width and instruction windows, future high-performance processors will likely cluster both the instruction window and functional units.

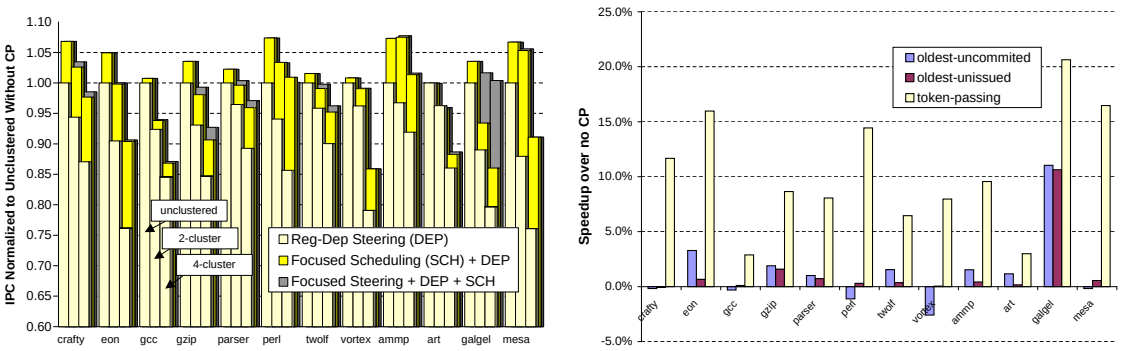
Clustering introduces two primary performance challenges. The first is the *latency to bypass* a result from the output of a functional unit in one cluster to the input of a functional unit in a different cluster. This latency is likely to be increasingly significant as wire delays worsen [69]. If this latency occurs for an instruction on the critical path, it will add directly to execution time.

The second potential for performance loss is due to increased *functional unit contention*. Since each cluster has a smaller issue width, imperfect instruction load balancing can cause instructions to wait for a functional unit longer than in an unclustered design. If the instruction forced to wait is on the critical path, the contention will translate directly to an increase in execution time.

Furthermore, steering policies have conflicting goals in that a scheme that provides good load balance may do a poor job at minimizing the effect of inter-cluster bypass latency.

The critical path can mitigate both of these performance problems. First, to reduce the effect of inter-cluster bypass latency, we perform *focused instruction steering*. The goal is to incur the inter-cluster bypass latency for non-critical (as opposed to critical) instructions where performance is less likely to be impacted. The baseline instruction steering algorithm for our experiments is the industry-standard *register-dependence* heuristic. This heuristic assigns an incoming instruction to the cluster that will produce one of its operands. If more than one cluster will produce an operand for the instruction (a *tie*), the producing cluster with the fewest instructions is chosen. If all producer instructions have finished execution, a load balancing policy is used where the incoming instruction is assigned to the cluster with the fewest instructions. In comparison to previous work, this policy is similar to the scheme used by the highly regarded distributed instruction window work of Palacharla *et al.* [69]. Our *focused instruction steering* optimization improves the baseline heuristic in how it handles ties: if a tied instruction is critical, it is placed into the cluster of its critical predecessor. This optimization was also performed by Tune *et al.* [110].

Second, to reduce the effect of functional unit contention, we evaluated *focused instruction scheduling*, where critical instructions are scheduled for execution before non-critical instructions. The goal is to add contention only to non-critical instructions, since they are less likely to degrade performance. The oldest-first scheduling policy is used to prioritize among critical instructions, but our experiments found this policy does not have much impact due to the small number of critical instructions. The baseline instruction scheduling algorithm gives priority to *long latency* instructions. Our experiments found this heuristic performed slightly better than the *oldest-first*



(a) Scheduling in clustered architectures. **(b)** Comparison to heuristics-based predictors.

Figure 7.1: **Critical path scheduling decreases the penalty of clustering. (a)** The token-passing predictor improves instruction scheduling in clustered architectures (8-way unclustered; two 4-way clusters; and four 2-way clusters are shown). As the number of clusters increases, critical-path scheduling becomes more effective. **(b)** Results for four 2-way clusters using both *focused instruction scheduling* and *steering* shows that the heuristic-based predictors are less effective than the token-passing predictor.

scheduling policy.

Experiments. The improvements due to *focused instruction scheduling* and *focused instruction steering* are shown in Figure 7.1(a) for three organizations of an 8-way issue machine: unclustered, two clusters, and four clusters. The execution time is normalized to the baseline machine (unclustered without any focused optimizations). We find that:

- On an unclustered organization, the critical path-based policy produces a speedup of as much as 7% (3.5% on average).
- On a 2-cluster organization, the critical path turns an average slowdown of 7% to a small *speedup* of 1% over the baseline. This is a speedup of up to 17% (7% on average) over register-dependence steering alone.
- On a 4-cluster organization, the critical path reduces performance degradation from 19% to a much more tolerable 6% degradation. Measured as speed up over register-dependence

steering, we improve performance by up to 21% (10% on average).

From these results, we see that the token-passing predictor is increasingly effective as the number of clusters increases. This is an important result considering that technological trends may necessitate an aggressive next-generation microprocessor, such as the one we model, to be heavily partitioned in order to meet clock cycle goals [2].

From Figure 7.1(a) we also see that *focused instruction scheduling* provides most of the benefit. We believe this is because *focused instruction steering* uses the critical path only to break ties, which occur in the register-dependence steering heuristic infrequently. Nonetheless, a few benchmarks do gain significantly from the enhanced steering, *e.g.,, gzip* gains 3% and *galgel* gains 14%.

Comparison to Prior Work. An alternative to *focused instruction scheduling* is to use a steering policy that prevents load imbalance that might lead to excessive functional unit contention, decreasing the importance of instruction scheduling within a cluster. We implemented several such policies, including the best performing non-adaptive heuristic (MOD3) studied by Baniasadi and Moshovos [10]. MOD3 allocates instructions to clusters in a round-robin fashion, three instructions at a time. While these schemes sometimes performed better than register-dependence steering, register-dependence performed better on average in our experiments. Most importantly, register-dependence steering with focused instruction scheduling *always* performed better (typically much better) than MOD3.

In Figure 7.1(b), we compare the token-passing predictor to the two heuristics-based predictors described in Section 6.1.2 (oldest-uncommitted and oldest-unissued) performing both *focused instruction scheduling* and *focused instruction steering* on a 4-cluster organization. Clearly,

neither heuristics-based predictor is consistently effective, and they even degrade performance for some benchmarks (*e.g.*,, for *vortex*, *perl*, and *crafty*). Our conjecture is that instruction scheduling optimizations require higher precision than heuristics can offer.

Note that even for *galgel*, where the oldest-unissued scheme compared favorably to the token-passing predictor in Section 6.1.2, Figure 7.1(b), the token-passing predictor produces a larger speedup. Upon further examination, we found that (across the benchmarks) the oldest-unissued predictor's accuracy degrades significantly after *focused instruction scheduling* is applied. This may be due to the oldest-unissued predictor's inherent reliance on the order of instructions in the instruction window. Since scheduling critical instructions first changes the order of issue such that critical instructions are unlikely to be the oldest, the predictor's performance may degrade as the optimization is applied. In general, a predictor based on an explicit model of the critical path, rather than on an artifact of the microexecution, is less likely to experience this sort of interference with a particular optimization.

In summary, it is worth noting that the significant improvements seen for scheduling execution resources speak well for applying criticality to scheduling other scarce resources, such as ports on predictor structures or bus bandwidth. In general, the critical path can be used for intelligent resource arbitration whenever a resource is contended by multiple instructions. The multipurpose nature of a critical-path predictor can enable a large performance gain from the aggregate benefit of many such simple optimizations.

Criticality in a Non-uniform ILDP Architecture

In this section, we evaluate the success of slack in guiding a non-uniform control policy. Since the design of the underlying machine is largely dependent upon the characteristics of the

workload, we start our exploration with a characterization of the slack available in our benchmark suite. We use this characterization to define an aggressively non-uniform (power-aware) microarchitecture whose non-uniformities can be effectively hidden by employing a slack-based control policy. We compare our slack-based policy with several policies based on existing control techniques and discover that slack is remarkably more successful at hiding the performance penalties that arise due to non-uniform resources.

Slack Characterization. As discussed in Chapter 4, there are three notions of slack that might be relevant to non-uniform architecture design: local, global, and apportioned. Local slack is the easiest to measure in hardware, since it involves simply measuring the difference in arrival times of two events at a node (*e.g.*, for the instruction $ADD \leftarrow R1, R2$, how many cycles sooner was $R1$ ready than $R2$?). In contrast, global and apportioned slack involve a propagation-style analysis, for which we would employ the token-passing analyzer. So, one of the goals of the characterization will be to determine if local slack is sufficient for driving control policies. Another will be to gain information as to how to design a non-uniform microarchitecture to match the needs of the workload. For simplicity, we only look at the slack characteristics of E nodes for this study.

To start off, Figures 7.2(a)-7.2(c) plot the local, global and apportioned slack found in *gcc*, *gzip*, and *perl*, respectively. These three benchmarks were chosen because they illustrate the two extreme results (*gcc* and *gzip*) and a typical result (*perl*) from the full set of measurements we performed.

Local and global slack. The slack measurements reported in the charts should be interpreted as follows: for each data point (**x**, **y**), **y**% of (dynamic) instructions have **x** *or more* cycles of slack. In *gcc*, for instance, approximately 36% of instructions have local slack of five or more cy-

cles. In general, we observe that relatively few instructions contain local slack that is large enough to be exploitable: on average only about 20% of instructions have local slack of five or more cycles. At the same time, we notice that a small number of instructions contain extremely large local slack (in *gzip*, about 2% of instructions have more than 80 cycles of local slack). This large local slack is promising because a single instruction is unlikely to be able to exploit it all, allowing us to apportion it to instructions without enough local slack.

Note that, while the figures only show local slack for the *execution* of instructions (*E* nodes in our model), other micro-operations associated with an instruction may also exhibit local slack. For instance, we may be able to delay the *commit* of an instruction (represented by *C* nodes in our model) without delaying any other instructions. Since our dependence-graph model accounts for this commit micro-operation, we can also apportion this local slack to other instructions.

To determine to what extent large local slacks can be used by neighboring instructions, we examine global slack. Since the global slack of an instruction is the accumulation of all local slacks that could be "stolen" from other instructions, observing a lot of global slack on many instructions would speak well for the potential for exploitation, since this would mean that lots of local slack is "freely movable" across the microexecution. Indeed, this is the case: about 40% of instructions have more than 50 cycles of global slack. The key question now is what fraction of this global slack remains if we spread it out across neighboring instructions. We answer this question using apportioned slack.

Apportioned slack. To calculate apportioned slack, we must first decide on the apportioning strategy. Let us first consider giving **x** cycles of slack to as many instructions as possible. The amount of such apportioned slack is shown along with local and global slack in Figures 7.2(a)-7.2(c)

for a range of values of **x**.

Again, the experiments present good news: not only does the microexecution contain a lot of apportionable local slack (which we knew from global slack measurements), but this slack is also able to satisfy many instructions: on average, 75% of instructions can be apportioned slack of five cycles. Even in the least slackful benchmark, *gzip*, there are 64% of instructions that have 5 cycles of slack. This means, for instance, that most instructions can tolerate long-latency communication across a chip without hurting performance—as long as the delayed instructions are chosen wisely (*i.e*, with a good slack predictor and a good policy).

Of course, the above apportioning strategy does not reflect all the non-uniformities that a control policy may have to tolerate. For instance, another interesting question is how many loads can tolerate a long latency to the L1 data cache, a concern of wire-constrained designs such as the Grid Architecture [86]. To maximize slack on loads, we modify the above apportioning strategy such that no slack is apportioned to non-load instructions. Figure 7.2(d) reports the results of such an apportioning. We see that a remarkable number of loads could tolerate a long-latency L1 data cache hit. Namely, there are more than 65% of load instructions with a slack of 12 cycles, enough to tolerate an L2 hit. Together, the data suggest an opportunity to build selective L1-cache bypasses.

Breakdown of slack per opcode. In Figure 7.3, we examine how much apportioned slack is available to instructions of various types. The figure computes the breakdown for the two apportioning strategies described in Section 4.3.2: five-cycles-per-instruction and latency-plus-one-cycle. The figure classifies instructions into four categories: loads, stores, integer operations, and floating-point operations. (Note that our simulator discards all NOP instructions after fetch, and, thus, they are not included in any of the slack measurements.)

Figure 7.3 leads to several conclusions about what types of non-uniformities can be tolerated with slack.

- Most instructions (on average, greater than 75%) have enough slack to tolerate doubling their latency. This means we can run most functional units at half-speed without losing performance, provided we are successful at predicting which instructions have slack. This result is good news for the fast/slow pipelines microarchitecture we study in the next section.
- A large percentage of instructions of each type can have their latency doubled; this holds even for longer latency floating-point operations.
- There is no instruction type which nearly always has slack. Thus, a machine design that simply makes all functional units of a particular type slower is likely to degrade performance.

The Non-uniform Architecture. Based on the conclusions of the slack characterization, we came up with microarchitecture pictured in Figure 7.4. In this design, the microarchitecture is divided into two *pipelines*, with each pipeline consisting of half of the *instruction window, issue logic, and functional units*; and a copy of the register file. The design saves power by running one pipeline at half frequency, exploiting the (approximate) relationship $P \propto FV^2$ between power P, voltage V and frequency F. By halving the frequency, we can reduce voltage enough that the overall power consumption is reduced roughly to a fourth ($P \propto F^2$). (Note that reducing the frequency of such a large portion of the pipeline is a more aggressive power-aware design than one that only reduces the speed of the functional units.)

We find that by employing a slack-based control policy, we can keep performance loss due to reducing the frequency of one cluster to 3-4%, which is many times better than the best

non-slack based scheme. Furthermore, if we are willing to spend more area, we can add an extra slow cluster to completely eliminate any performance degradation while still maintaining most of the power benefits.

Control Policies At a first glance, it may seem that reducing the frequency on one pipeline introduces only one kind of non-uniformity. The reality is that in our design we need to deal with three forms of non-uniformity:

1. The *execution latencies* of functional units in the slow pipeline will be twice as large as those in the fast pipeline.

2. The *bypass latency* between the two pipelines will be longer than the intra-pipeline bypass latency, due to not only the physical distance but also due to crossing voltage domains.

3. The *effective issue bandwidth* of the slow pipeline will be half of the bandwidth of the fast pipeline, because the slow pipeline issues instructions every other fast cycle. This reduction in issue bandwidth manifests itself as increased contention (which happens to be the hardest constraint to deal with).

The important consequence of the third point is that frequency reduction reduces the effective bandwidth of the *entire* machine. This observation is important because it sets the correct expectation on the control policy: when a workload is bandwidth-limited (i.e., exhibits high IPC rate), no control policy will be able to avoid the performance penalty.

To attack the above three non-uniformities, we design a slack-based policy that controls two machine aspects:

- *Instruction steering*, which determines into which pipeline a dynamic instruction is sent.

- *Instruction scheduling*, which determines which of the data-ready instructions in a pipeline are executed.

We assume that the steering decision is performed before any scheduling decisions are carried out.

Our slack-based policy employs four bins, as introduced and motivated in Section 6.2. These four bins control to which pipeline an instruction will be steered, and also how the instruction will be scheduled within the pipeline (see Table 7.3). Note that we also experimented with two-bin policies (which performed steering but no slack-based scheduling), but the four-bin scheme performed up to 5% better.

To assign a slack bin to each static instruction, our slack policy uses a 4K-entry array of 6-bit saturating counters, indexed by PC. The counter is decremented by one if the slack sampling (see Section 6.2) detects that the instruction can tolerate a given pipeline and a given scheduling policy (i.e., is slackful enough for the pipeline/scheduling combination). The instruction is moved to a lower-numbered bin when the counter reaches zero and to a higher-bin if it is detected that it does not have enough slack for the given bin.

For best performance, we need to maintain a relative balance of instructions in each bin. For the fast/slow clusters application, we want approximately a third to a half of the instructions to be sent to the slow cluster and the rest to the fast (considering the steady-state execution bandwidth provided by the slow cluster is one half that of the fast). Furthermore, we want a fairly small percentage of instructions to have high-priority and be scheduled first in each cluster. If too many are scheduled first, the benefit of the optimization is diminished.

To maintain the desired balance, the hysteresis is specially designed for each bin. In order to have fewer instructions reside in a particular bin, we use hysteresis to make it more difficult

Name	Policy
Reg-Dependence	Perform load balancing if one pipeline is four times as full as another. Otherwise, steer instruction to pipeline that will produce one or more of its inputs. Steer to least-filled pipeline if all operands are ready
Fast-first Window	Send instructions to the fast pipeline until its window becomes half full, then apply register-dependence steering.
Fast-first Ready	Send instructions to fast pipeline until there were more ready instructions then issue slots over the last 5 cycles. Then, apply register-dependence steering.

Table 7.2: **Baseline policies for controlling fast/slow pipeline microarchitecture.**

to transition into that bin. For the fast/slow clusters application, the hysteresis used is shown in Table 7.3.

To avoid extreme load imbalance between the two clusters (which happens when too many instructions are detected as slackful, overloading the slow cluster), our policy occasionally overrides the the slack-based steering to correct the imbalance. Load balancing is invoked under the following condition: If the slow instruction window contains four times as many instructions as the fast window, the incoming instruction is sent to the fast cluster. Load balancing never steers instructions to the slow cluster.

We compare our slack-based policy to several policies based on existing (non-slack-based) control techniques. While we experimented with many such policies, we only present three that performed best (see Table 7.2). The first is a simple register-dependence steering policy, while the other two "favor" the fast pipeline over the slow one in that instructions are steered to the fast pipeline until some condition is met. We also evaluate the use of the ALOLD criticality predictor from Tune, et al. [110], as a replacement for the token-passing criticality analyzer [34] in the slack

Slack bin #	Policy decisions	Hysteresis counter
4	Fast pipeline, high priority schedule	Initialize to 0 upon entering level. Increase by 8 if detected not slackful.
3	Fast pipeline, low priority schedule	Initialize to 63 upon entering level. Immediately go to level 4 if detected not slackful.
2	Slow pipeline, high priority schedule	Initialize to 63 upon entering level. Immediately go to level 3 if detected not slackful.
1	Slow pipeline, low priority schedule	Initialize to 63 upon entering level. Immediately go to level 2 if detected not slackful.

Table 7.3: **Hysteresis implementing the four slack bins.** Note: if the slow instruction window contains four times as many instructions as the fast pipeline, the slack-based steering decision is overridden, and the incoming instruction is sent to the fast pipeline. Such load balancing never sends instructions to the slow pipeline.

detector. (We also experimented with the QOLD criticality predictor from the same work [96, 110], but the ALOLD predictor performed considerably better in our context.)

Experimental Evaluation We evaluate the set of control policies on a machine with one 3-wide fast pipeline and one 3-wide slow pipeline (*3f+3s*). The results, presented in Figure 7.5, yield two overall conclusions. First, our slack-based policy performs better than any non-slack policy, by 10% on average. Second, using slack reduces the performance degradation (with respect to the high-power *3f+3f* configuration) from an average of 16% to only 3%.

It is interesting to observe the effect of replacing the token-passing detector with the ALOLD predictor: while ALOLD performs better than the non-slack schemes, degrading performance by 10%, it appears that the token-passing detector is needed to accurately measure slack.

In an attempt to recoup the small performance loss of *3f+3s*, we experimented with other configurations where issue bandwidth is made equal to *3f+3f* through the addition of another slow pipeline. In these equi-bandwidth configurations, we found that our slack-based policy actually slightly improved performance over *3f+3f*, while the non-slack policies significantly degraded it, by

12–15% on average. Specifically, the additional configurations explored are summarized below:

- *3f+3s+3s:* one 3-wide fast pipeline and two 3-wide slow pipelines. This configuration has the same issue bandwidth as the *3f+3f* but a larger effective instruction window. While the performance for all policies improved over *3f+3s*, the relative performance of the four policies remained roughly the same: our slack predictor actually improved performance by 1% (compared to the 3f+3f machine), while all other policies degraded it, by 12-15% on average.

- *Half_3f+3s+3s:* one 3-wide fast pipeline and two 3-wide slow pipelines, where the window size of each slow pipeline is halved. This configuration has the same issue bandwidth and effective window size as *3f+3f*. Across the policies, performance was 1-2% worse than for *3f+3s+3s*, indicating the increased effective window size does have some performance benefit. Most of the gains, however, come from increased issue bandwidth.

Power Savings While the focus of this work is to evaluate slack as a tool for designing control policies, it is interesting to estimate the power savings of our non-uniform machine configurations. While an accurate power analysis is beyond our scope, we will compute *asymptotic* savings. (*Actual* power savings will, in any case, depend upon the power contribution of the machine core (i.e., the instruction window, issue logic, register file, and functional units), which is highly dependent upon the particulars of the processor implementation).

To estimate the fraction of the *core* power that we save with each configuration, we can employ the quadratic relationship of frequency/voltage reduction to power (naturally, practical device considerations may change this ratio to some degree). Assuming that halving the frequency decreases the power consumption to a quarter, we have: in the 3f+3s configuration, we save

$100 - 50 + 50 * 1/4 = 37.5\%$ of the power of the core; and in the 3f+3s+3s configuration we save $100 - 50 + 50 * 1/4 + 50 * 1/4 = 25\%$.

To estimate savings to overall chip power, we employ a methodology similar to Bahar and Manne [8], which extrapolated the savings from available power estimates of real machines. Wilcox showed the 8-way issue Alpha 21464 was expected to have 66% of its power dissipation in its core [116]. Since we assumed a 6-way machine for this study, we estimate 50% of total chip power going to the core. Under this assumption, the 3f+3s configuration would reduce the overall chip power by approximately 18%, while the 3f+3s+3s machine saves approximately 12%.

Of course, our slack policy itself consumes some power, but we expect it to be a very small amount of overall chip power, for the following reasons: the criticality detector is a very simple hardware structure consisting of two small arrays, an array of size ROB_size $*3$ nodes per instruction $*8$ tokens $= 768$ bytes that is read and written during training and a 4KB array to store the slack bin predictions. Furthermore, the predictor can be used for hiding many different non-uniformities (as opposed to just the single optimization explored in this section) and, thus, its power dissipation may be amortized across numerous applications.

Other Resource Arbitration Applications

In this section, we will further illustrate the value of using criticality for resource arbitration by briefly discussing some applications developed by others.

Heterogeneous cache organization. Rakvic, *et al.* [75] propose introducing a *vital* cache, which is a small, fast-access cache that sits in front of the traditional L1 cache (inclusivity is maintained). Such a small cache cannot hold the data working set of most workloads. However, it is large enough

to hold most of the *critical* portion of the working set of many workloads. The estimate they use for identifying criticality is what we would call local slack. If a load has some local slack (*i.e.*, it's data is not consumed immediately), it is considered *non-vital* and not stored in the vital cache. They achieve an average of a 12% speedup with this cache organization. It may be possible to improve upon this result with a more global computation of criticality.

Instruction window utilization. Crowe, *et al.* [27] identified which instructions had enough slack such that they can tolerate some delay before being placed into the instruction window. These instructions are placed into a *deferred* queue, giving priority to the more critical instructions. They identify the slack of each static instruction through an offline analysis, where the static slack of an instruction is defined as the minimum global slack over all dynamic instances of that instruction. They achieved an 11% speedup over their baseline four-wide out-of-order processor.

7.2.2 Speculation Control

Another class of control-policy applications of criticality is speculation control. The goal of speculation control is to only make predictions when there is potential for performance improvement. Performance improvement is possible when the speculation attacks an event with positive cost, while it is not possible if that event has a positive slack. Reducing the number of useless predictions in this manner has two benefits: (1) fewer misspeculations and (2) less overhead due to the speculation.

It is easy to see how speculation control could reduce the number of misspeculations. For example, value prediction is only potentially useful for instructions that are on the critical path. If the instruction is off the critical path, predicting it's output will not help at all, since speeding up

the instruction's execution will not improve program performance. On the other hand, if we attempt a prediction for a noncritical instruction and are incorrect, performance will likely be substantially impacted due to the recovery cost.

Speculation control can also reduce overhead in situations where the speculation has a cost associated with it. One example is pre-execution, where a chain of dependent instructions leading up to a frequently cache-missing load (or frequently mispredicted branch) are processed early, well before the load or branch is encountered in program order. The benefit in pre-execution is that the load's data can be prefetched (or the branch outcome can be known) by the time they are normally encountered. The downside of this technique is that the pre-execution uses resources (functional units, window slots) that could be used for the normal program execution. So, it is logical that pre-execution should only be applied where there is potential for substantial benefit, something that criticality ($icost$ in particular) can indicate.

Reducing Misspeculations

Focused value prediction is an optimization that uses the critical path for reducing the frequency of (costly) misspeculations while maintaining the benefits of useful predictions. By predicting only critical instructions, we improved performance by as much as 5%, due to removing nearly half of all value misspeculations.

The Problem. Value prediction is a technique for breaking data-flow dependences and thus also shortening the critical path of a program [62]. In fact, the optimization is only effective when the dependences are on the critical path. Any value prediction made for *non*-critical dependences will not improve performance; even worse, if such a prediction is incorrect, it may severely degrade

performance. In *focused value prediction*, we only make predictions for critical path instructions, thus reducing the risk of misspeculation while maintaining the benefits of useful predictions.

Table Sizes	Context: 1st-level table: 64K entries, 2nd-level table: 64K entries, Stride: 64K entries. The tables form a hybrid predictor similar to the one in [113]
Confidence	4-bits, saturating: Increase by one if correct prediction, decrease by 7 if incorrect, perform speculation only if equal to 15 (This is similar to the mechanism used in [19]).
Mis-speculation Recovery	When an instruction is misspeculated, squash all instructions before it in the pipeline and re-fetch (like branch mispredictions.)

Table 7.4: **Value prediction configuration.**

Experiments. We used a hybrid context/stride predictor similar to the predictor of Wang and Franklin [113]. The value predictor configuration, detailed in Table 7.4, deserves two comments: In order to isolate the effect of value misspeculations from the effects of value-predictor aliasing, we used rather large value prediction tables. Second, while a more aggressive recovery mechanism than our squash-and-refetch policy might reduce the cost of misspeculations, it would also significantly increase the implementation cost. We performed experiments with focused value prediction on the seven benchmarks that our baseline value predictor could improve. We evaluate our token-passing predictor and the two heuristics predictors.

Figure 7.6(a) shows the number of misspeculations obtained with and without filtering predictions using the critical path. While the oldest-unissued heuristic eliminated the most misspeculations, it is clear from Figure 7.6(b) that it also eliminated many beneficial correct speculations. The more precise token-passing predictor consistently improves performance over the baseline value predictor and typically delivers more improvement than either heuristic. The absolute performance gain is modest because the powerful confidence mechanism in the baseline value predictor already

filters out most of the misspeculations. Nonetheless, the potential for using the critical path to improve speculation techniques via misspeculation reduction is illustrated by 5 times more effective value prediction for *perl* and 7–20% more effectiveness for the rest of the benchmarks.

Reducing Speculation Overhead

In addition to reducing the number of mispredictions, criticality can also be used to reduce the overhead caused by various forms of speculation. There are two principle ways the overhead reduction can be achieved. The first and most direct is to eliminate speculation when it is not possible that it could improve performance, as we did for the value prediction case study above. A second is to use slack analysis to "hide" the overhead behind other computation. For example, pre-execution requires fetching and executing a chain of instructions before they arrive in the normal program stream. If we could detect when there is fetch slack available, we would know when we could fetch the pre-execution stream without hurting program performance.

We have not studied the pre-execution application in detail, but there has been some related work that tackled part of the problem. Specifically, Petric and Roth [71] used criticality constructed from our graph model to identify cache misses most in need of being pre-executed, meaning that the data-dependence chain leading up to the miss is pushed forward in the program stream. The specific metric they use is simple *cost*, *i.e.*, the benefit that could be achieved by pre-executing a load (in isolation). Of course, since multiple loads are often being serviced simultaneously (and thus have very low individual costs), using simple cost without accounting for interactions would miss many optimization opportunities. Instead of explicitly measuring interaction costs, they average two values to obtain an adjusted cost metric: (1) the cost of the load as we define it and (2) a more optimistic cost value assuming all other loads have been successfully prefetched. By

employing this metric as part of their overall pre-execution system, they were able to avoid some unnecessary work, both saving energy and increasing performance.

Note that Petric and Roth's work improves pre-execution by avoiding unnecessary work; it does not attempt to use criticality to hide the overhead of the pre-execution threads. It may be possible to use criticality (especially for fetch nodes) to identify when the main thread can afford to lose some bandwidth without performance loss. No work has yet attempted this optimization, however.

7.2.3 Dynamic Hardware Reconfiguration

One of the techniques that architects have explored for reducing energy dissipation is to change the hardware configuration dynamically to match the program's needs. There are two general strategies. The first is to resize hardware structures such that the machine is more "balanced" or matched to the needs of the program. In other words, no resizable resource has slack in that it could be reduced in size without impacting performance. Since for many structures, such as the instruction window, there is a quadratic relationship between their size and their energy dissipation, resizing has the potential to help substantially. The second strategy, which is more popular in industry, is to dynamically adjust the frequency to meet the program's needs. Since frequency also has a quadratic relationship with energy, reducing frequency can have a very substantial benefit.

Structure Resizing. Criticality is very directly applicable to the problem of resizing hardware structures. Consider the problem of resizing the instruction window. The instruction window constraint is represented by CD edges in the graph model. If the CD edge is critical, the (limited) size of the instruction window is impacting performance. The inverse is also true, so a non-critical

CD edge means the instruction window size can be reduced without hurting performance. This approach can be applied for resizing any hardware resource that is modeled in the graph.

Some work by Sasanka, *et al.* [87] developed a specialized hardware cost estimator for making resizing decisions. The design of their estimator was inspired by our last-arriving edge criticality analyzer, but, by specializing it to the task of resizing the instruction window, it is made less expensive to implement. We compared their cost estimator to our more general criticality detector and found that both perform similarly well. Their criticality-based policy saved slightly more energy than the state-of-the-art policies at the time.

Frequency Scaling. The most advanced proposals for frequency scaling divide the chip real estate into multiple clock domains, each with an independently controllable frequency. The goal with this sort of architecture is to set each zone's frequency as low as possible while still maintaining good performance. Semeraro, *et al.* [95] used an offline microarchitecture graph-based slack analysis to determine good frequencies for different segments of code. Marculescu [64] furthered this work to include dynamic criticality information, concluding that a hybrid between our token-passing analyzer and the heuristic predictors provide the best energy/performance tradeoff for MCDs. In a similar work, Chin, *et al.* [21] used slack to control the frequencies of different pipelines in a clustered architecture.

7.3 Hardware Design Help

The typical approach architects use to better understand how machine alterations affect performance is to run many simulations with many of the different configurations under consideration. The execution time reported by the simulations provide insight into what configurations will

likely perform best in practice. To gain more fine-grain understanding of why performance is as it is, architects will examine counters. For example, if the cache miss count increases when a simulation is run with a larger instruction window, one might logically conclude that there is an interaction between the window and cache misses.

Our work on criticality can improve the state of hardware design in three ways:

- *Early Results.* When exploring new architectural ideas, it is typically very time consuming to write an entire simulator to test their performance. A quicker and easier alternative is to alter the graph model to reflect the new architecture. A performance estimate can then be obtained by measuring critical path lengths of instances of the new model. This methodology can enable architects to quickly explore a larger portion of the design space than would otherwise be practical.

- *Test more configurations quickly.* A related advantage is the ability to test out many configurations very quickly. For example, testing the cross product of all possible instruction window sizes, fetch bandwidths, issue bandwidths, and load-store queue sizes results in an exponential blowup in the number of simulations required. Testing these configurations on the graph still results in an exponential blowup, but each graph analysis is much, much faster than running a simulation — resulting in the ability to explore a larger portion of the design space.

- *Costs, Interactions, and Performance Breakdowns.* A final application of criticality, which we have spent the most time exploring, is a better alternative to performance counters for gaining insight into *why* a particular architecture performs the way that it does. This problem is important since it can lead to new insights quickly. In fact, the methodology can provide

interpretations of performance (almost) automatically; these interpretations would otherwise require substantial effort from an experienced architect.

Our work has focused on developing the alternative to performance counters (the last bullet). Below we present a case study that illustrates how interaction costs and performance breakdowns can be used to gain insights into performance.

7.3.1 Icost Tutorial: Optimizing a long pipeline

Several recent studies have found significant performance improvements possible by increasing the length of the processor pipeline. The improvement comes from increased clock frequency, but this improvement is unfortunately offset by the increasing latency of *performance-critical loops*. A loop is a feedback path in the pipeline, where the result of one stage is needed by an earlier stage. Three of the most critical loops include: (i) the latency of a level-one data cache access, (ii) the latency to issue back-to-back operations (the issue-wakeup loop), and (iii) branch mispredictions [101, 53, 47, 13].

In this section, we present a tutorial on how interaction costs can help architects during design of a new processor. Specifically, interaction costs can show us how to mitigate the performance impact of critical loops in processors with long pipelines. Finally, we compare our icost analysis conclusions to those of a conventional sensitivity study.

The level-one data cache access loop

Let's assume that the circuit designers optimized the level-one data cache access as much as possible, but nonetheless the latency was higher than expected, say four cycles instead of the

typical one or two. The question now is: What is the *most effective* way to change the microarchitecture to mitigate the effect of the high latency? Would it help to: (a) enlarge the branch predictor; (b) increase the number of load ports; (c) increase the data cache size; or (d) increase the fetch bandwidth? Certainly these changes will reduce the cost of each of these resources (if they were on the critical path), but will they also reduce the cost of data cache accesses?

What we are looking for is a choice of something other than data accesses to optimize that will indirectly reduce the cost of those accesses. Optimizing some resource such as fetch bandwidth certainly will not affect the latency of data accesses, but the optimization might cause some of the latency to be removed from the critical path (or, in other words, "hidden" or "tolerated" by the machine). In essence, we are looking for *serial* interactions, since any resource that serially interacts with data accesses provides us an alternative resource for optimization that will enable us to remove the same set of cycles.

In our case study, before computing the interaction costs, we hypothesized what the outcome of the analysis could be, which amounted to predictions of where *serial* interactions would occur. We thought data dependences between data-cache missing loads or ALU operations and level-one data-cache accesses might cause such a serial interaction. Another possibility would be an interaction between branch mispredicts and data-cache accesses, since loads often feed branches.

The results of the analysis is shown in Table 7.5 (simulator parameters are in Table 7.1 in Section 6.3.5). For brevity, the breakdown presents only those interaction costs that involve data-cache accesses, labeled 'dl1' in the table. In total, there would be $2^8 - 1 = 255$ costs and interaction costs if all of them were shown.

Before examining the correctness of our hypotheses, let's attempt to gauge the importance

Category	bzip	crafty	eon	gap	gcc	gzip
dl1	**22.2**	**24.2**	**18.2**	**13.5**	**18.3**	**30.5**
win	16.4	15.1	15.7	41.0	13.6	23.0
bw	4.4	8.0	7.7	2.8	8.2	5.7
bmisp	41.0	28.6	15.8	12.3	26.3	25.8
dmiss	23.8	7.1	0.7	23.5	26.3	7.7
shortalu	9.9	11.4	5.4	13.8	5.1	20.4
longalu	0.3	0.9	11.8	5.6	0.4	0.7
imiss	0.0	0.7	7.8	0.7	2.2	0.1
dl1+win	**-5.2**	**-10.5**	**-6.8**	**-6.0**	**-4.2**	**-15.3**
dl1+bw	5.6	9.9	8.1	2.8	10.0	6.0
dl1+bmisp	**-10.8**	**-5.4**	**-4.9**	**-2.9**	**-7.0**	**-3.4**
dl1+dmiss	-0.7	-1.2	-0.4	-0.4	-1.4	-0.4
dl1+shortalu	**-4.1**	**-4.3**	**-1.0**	**-0.2**	**-1.6**	**-8.2**
dl1+longalu	-0.3	0.1	-0.3	0.1	-0.3	-0.4
dl1+imiss	0.0	0.0	0.8	0.1	0.3	0.0
Other	-2.5	15.4	21.4	-6.7	3.8	7.8
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Category	mcf	parser	perl	twolf	vortex	vpr
dl1	**7.7**	**19.0**	**31.6**	**19.4**	**28.8**	**19.7**
win	4.2	17.3	4.4	25.1	47.1	23.2
bw	0.5	2.9	8.6	3.9	5.3	5.8
bmisp	26.9	16.5	38.0	24.1	1.9	24.9
dmiss	81.0	32.9	1.4	34.4	21.8	33.7
shortalu	1.4	19.7	7.3	7.8	4.9	7.6
longalu	0.0	0.1	0.8	4.2	1.6	3.6
imiss	0.0	0.1	5.2	0.0	2.8	0.0
dl1+win	**-0.2**	**-6.1**	**-4.3**	**-4.1**	**-27.6**	**-5.7**
dl1+bw	0.3	4.9	9.6	1.5	17.6	1.8
dl1+bmisp	**-2.4**	**-2.8**	**-7.6**	**-6.5**	**-0.2**	**-4.6**
dl1+dmiss	-0.5	-1.4	-0.2	-1.3	-1.8	-2.5
dl1+shortalu	**-0.1**	**-3.6**	**-1.4**	**-0.3**	**-4.0**	**-1.3**
dl1+longalu	0.0	-0.0	-0.7	0.0	-1.3	-0.3
dl1+imiss	0.0	0.0	1.0	0.0	0.4	0.0
Other	-18.8	0.5	6.3	-8.2	2.7	-5.9
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Table 7.5: **Breakdowns for optimizing a long pipeline: Four-cycle level-one cache.** Interaction costs are presented here as a percent of execution time and were calculated using the dependence graph in a simulator. The categories are: 'dl1' → level-one data cache latency; 'win' → instruction window stalls; 'bw' → processor bandwidth (fetch,issue,commit bandwidths); 'bmisp' → branch mispredictions; 'dmiss' → data-cache misses; 'shalu' → one-cycle integer operations; 'lgalu' → multi-cycle integer and floating-point operations; and 'imiss' → instruction cache misses. Note that 'Other', denoting the sum of all interaction costs not displayed, can be negative since the interaction costs can be negative.

of interactions in general. If we sum up the singleton costs, say for $crafty$, we get a very high value, $24.5 + 16.3 + 6.0 + 16.4 + 6.7 + 11.3 + 0.8 + 0.6 = 92.6\%$. Does this mean interactions are only important for a small portion of the execution time, *e.g.*, 7.4% for crafty? The answer is "no", since these singleton costs could be counting the same cycles multiple times — in other words, serial interactions (negative icosts) may exist. In fact, the sum of the singleton costs for $vortex$ is over 100, at 104%, which is only explainable by serial interactions. As expected, $vortex$ does have interactions (in fact, large ones), both parallel and serial (and this is seen even when only considering interactions including $dl1$). So, we cannot make conclusions on the importance of interactions by looking at singleton costs alone.

In analyzing the data, notice first that data-cache accesses have a large singleton cost, typically contributing 15–25% of the execution time. This means that 15–25% of the execution time would be eliminated if the data-cache access latency was reduced to zero. As for the interactions, we see that some of our hypotheses were correct: for instance, there are significant serial interactions between data-cache accesses and ALU operations (*dl1+shalu*), suggesting we could mitigate the long data-cache loop by reducing ALU latency (perhaps through value prediction [63, 19] or instruction reuse [99]).

However, other conclusions from the analysis were not predicted beforehand. For example, it was hypothesized that large serial interaction might exist between data-cache misses and data-cache accesses. In reality, this interaction is very small: reducing data-cache misses is unlikely to mitigate the effect of the high latency data-cache loop.

We also see that the largest serial interaction for most benchmarks is with instruction window stalls. Thus, perhaps the most effective mitigation of the data-cache loop would be to increase

the size of the instruction window — a result that may be difficult to predict before performing the analysis.

Also, note that the *magnitude* of the interactions vary significantly across benchmarks. This variability suggests that interaction costs could be useful in workload characterization: their magnitude gives a designer early insights into what optimizations would be most suitable for the most important workloads.

Balanced machine design. One particularly interesting interpretation of interaction costs enables microarchitects to determine how *balanced* a machine design is, as well as determine where the imbalances exist.

We start with a definition of "balanced". A machine is said to be balanced if no processor resource can be reduced in size or made slower without impacting execution time. In other words, there is no wasted effort (slack) in any stage of any instruction's processing. In terms of the graph, all paths are of equal length — hence, there is no dominant critical path.

Interaction cost makes it easy to determine if a machine is balanced. Consider the icost between two resources, *e.g.*, data-cache accesses (*dl1*) and the instruction window (*win*). The machine is balanced with respect to these two resources if and only if the individual cost of each of the resources is zero ($cost(dl1) = cost(win) = 0$); and, thus, all of the cycles for which the resources are responsible are contained in the (non-negative) interaction cost between the resources ($icost(dl1, win) \geq 0$). As an example, consider the effect on costs and interaction costs when increasing the size of the window from 64 to 256, presented for $vortex$ below.

	vortex		
	64	128	256
dl1	28.5	9.8	4.3
win	39.4	21.3	13.6
dl1+win	**-25.9**	**-8.14**	**-2.7**
Exe Time	*100.0*	*80.8*	*75.0*

Notice how increasing the window size *reduces* the cost of the individual resources but *increases* the size of the interaction. (In this case, increasing the icost causes it to become less negative.) From this observation, we know the critical path is less dominant when the window size is larger — *i.e.*, the parallel paths are closer in length to the critical path, reducing the magnitude of the serial interaction. In other words, the machine is more balanced when the window is larger.

To generalize, for any set of resources, a larger icost (less negative or more positive) implies a more balanced machine design. A machine can be said to be completely balanced when all of the execution time exists as a parallel interaction among all of the processor resources. For example, if the machine has three resources (A, B, and C), the machine is balanced if and only if all of the individual costs and lower-term interactions are zero ($cost(A) = cost(B) = cost(C) = icost(A,B) = icost(B,C) = icost(A,C) = 0$) while the highest-term interaction is equal to the execution time ($icost(A,B,C) = exe.time$). Large individual costs and significant serial interactions indicate where the imbalances exist.

The issue-wakeup loop

Suppose that a long pipeline demanded a two-cycle issue-wakeup latency, instead of the typical one. This will, of course, reduce performance, since ALU operations will not be able to issue back-to-back. Can we use serial interactions to determine how to mitigate the performance loss?

From the breakdown of Table 7.6, we see significant serial interactions between ALU operations and several event classes: window stalls, branch mispredicts, and level-one cache accesses. The most significant interaction is, again, with window stalls; it is as large as -24% for *gap*. Because of this serial interaction, increasing the window size is more beneficial when the issue-wakeup latency is higher. For instance, we found that the speedup for *gap* when the window size is increased from 64 to 128 is 12% if the issue-wakeup latency is one and 18% if the latency is two, a difference of 50%.

The negative interaction costs also reveal for which benchmarks it is *not* going to be possible to mitigate the effect of longer pipeline loops by optimizing other parts of the machine. This is the situation in *gcc*, which exhibits very little serial interaction.

The branch misprediction loop

Finally, we consider the branch misprediction loop. Can we modify the microarchitecture to reduce branch misprediction costs? How about increasing the window size? Will that work to reduce branch misprediction loop cost in the same way it did for the other two loops?

The interaction costs in Table 7.7 reveal that the answer is no. Instead of a serial interaction between branch mispredictions and window stalls, there is a *parallel* interaction. This parallel interaction tells us there are a significant number of cycles that can be eliminated only by optimizing *both* classes of events simultaneously, *i.e.*, by optimizing both branch mispredictions and window size. In other words, reducing window stalls alone is not likely to significantly reduce branch misprediction costs.

For a couple of benchmarks, *mcf* and *parser*, we do see significant serial interactions with data cache misses (dmiss), however. In particular, for *mcf*, the serial interaction of

	bzip	crafty	eon	gap	gcc	gzip
shortalu	**19.7**	**25.6**	**13.2**	**39.7**	**18.9**	**40.0**
win	22.9	21.1	18.5	51.4	9.3	32.3
bw	2.9	5.7	8.0	1.5	6.2	3.4
bmisp	34.5	25.1	15.4	8.0	39.9	20.3
dmiss	28.3	14.5	4.1	16.5	12.7	17.9
dl1	12.2	12.1	10.0	4.6	16.1	15.4
imiss	0.4	5.4	9.4	2.8	8.9	0.4
longalu	0.2	0.7	12.5	4.4	0.5	0.6
shortalu+win	**-4.7**	**-11.9**	**-3.8**	**-32.0**	**-3.3**	**-19.7**
shortalu+bw	8.0	7.4	3.7	6.4	4.0	4.9
shortalu+bmisp	**-9.6**	**-5.8**	**-1.6**	**1.1**	**-8.7**	**-4.6**
shortalu+dmiss	-0.4	-0.2	0.1	1.9	0.1	-0.8
shortalu+dl1	**-5.9**	**-5.6**	**-2.2**	**-1.2**	**-5.4**	**-8.2**
shortalu+imiss	0.0	0.1	0.2	0.1	0.5	0.0
shortalu+longalu	-0.1	-0.6	0.4	-2.0	-0.3	-0.4
Other	-8.4	6.4	12.1	-3.2	0.6	-1.5
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

	mcf	parser	perl	twolf	vortex	vpr
shortalu	**3.1**	**12.0**	**17.7**	**17.5**	**13.2**	**14.8**
win	3.5	11.8	4.6	27.4	39.5	26.7
bw	0.3	3.3	8.1	2.1	5.9	3.9
bmisp	24.9	23.1	38.8	20.5	2.5	22.3
dmiss	83.6	49.2	4.7	43.7	24.0	46.7
dl1	4.5	11.8	18.2	8.6	17.1	8.7
imiss	0.0	0.3	9.1	0.8	11.4	0.4
longalu	0.0	0.0	0.7	3.8	1.1	2.8
shortalu+win	**0.0**	**-4.3**	**-3.4**	**-1.4**	**-12.2**	**-3.2**
shortalu+bw	0.4	3.9	6.0	2.7	4.4	1.6
shortalu+bmisp	**-2.0**	**-1.6**	**-4.7**	**-4.8**	**-0.4**	**-5.6**
shortalu+dmiss	0.2	0.1	0.0	0.9	-0.3	0.2
shortalu+dl1	**-0.3**	**-4.3**	**-4.0**	**-0.7**	**-10.4**	**-1.3**
shortalu+imiss	0.0	0.0	0.4	0.0	0.2	0.0
shortalu+longalu	-0.0	-0.0	-0.1	-0.9	0.5	0.0
Other	-18.2	-5.3	3.9	-20.2	3.5	-18.0
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Table 7.6: **Breakdowns for optimizing a long pipeline: Two-cycle issue-wakeup loop.**

	bzip	crafty	eon	gap	gcc	gzip
bmisp	**34.0**	**26.6**	**15.6**	**11.7**	**39.8**	**23.5**
dl1	10.6	11.6	9.4	6.7	15.5	18.2
win	22.0	15.4	16.3	40.4	7.8	28.8
bw	6.0	9.6	10.7	3.4	8.3	5.8
dmiss	32.2	17.4	4.6	25.0	14.2	22.5
shortalu	8.5	11.8	4.5	17.5	9.3	23.1
longalu	0.3	0.8	14.2	5.5	0.5	0.7
imiss	0.5	6.5	11.8	3.9	10.7	0.5
bmisp+dl1	-6.6	-3.5	-3.7	-1.7	-8.3	-1.9
bmisp+win	**10.2**	**10.8**	**7.0**	**7.8**	**11.8**	**8.3**
bmisp+bw	-1.6	-2.5	-2.3	-1.1	-2.4	-1.5
bmisp+dmiss	**-2.4**	**0.5**	**-0.2**	**0.2**	**-0.7**	**-0.9**
bmisp+shortalu	-5.6	-3.7	-0.7	0.6	-5.0	-3.3
bmisp+longalu	-0.0	0.2	-0.5	0.3	0.0	0.1
bmisp+imiss	-0.0	-0.2	-1.6	-0.4	-0.5	0.0
Other	-8.1	-1.3	14.9	-19.8	-1.0	-23.9
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

	mcf	parser	perl	twolf	vortex	vpr
bmisp	**24.2**	**24.3**	**40.3**	**19.8**	**2.5**	**21.4**
dl1	4.5	10.1	17.5	9.8	11.9	9.5
win	3.6	9.7	2.2	30.0	34.4	27.0
bw	0.4	5.2	11.6	3.6	8.5	5.2
dmiss	85.2	53.2	5.2	49.5	26.0	51.1
shortalu	1.4	5.0	8.2	8.2	5.4	7.4
longalu	0.0	0.0	0.7	4.1	1.2	3.3
imiss	0.0	0.3	10.8	0.8	13.0	0.5
bmisp+dl1	-1.2	-3.2	-6.7	-3.7	-0.3	-2.3
bmisp+win	**54.4**	**39.3**	**12.3**	**31.9**	**2.1**	**31.6**
bmisp+bw	-0.2	-2.1	-5.5	-1.1	-0.1	-2.3
bmisp+dmiss	**-14.6**	**-6.0**	**-0.7**	**-0.6**	**-0.1**	**-2.1**
bmisp+shortalu	-0.9	-1.1	-2.6	-2.3	-0.2	-3.1
bmisp+longalu	0.0	0.0	0.7	1.1	0.0	1.4
bmisp+imiss	0.0	0.0	-1.2	0.0	-0.3	0.0
Other	-56.8	-34.7	7.2	-51.1	-4.0	-48.6
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Table 7.7: **Breakdowns for optimizing a long pipeline: 15-cycle branch mispredict loop.**

-16.2% tells us that as much as 60% of the cost of branch mispredictions (16.2/26.5*100%) could be eliminated through optimization of data cache misses. Intuitively, this effect is likely due to cache-missing loads providing data that is used to determine a branch direction. Again, interaction costs help: we can quantify the importance of this effect for particular workloads, even determining the static instructions where it occurs, helping to guide prefetch optimizations.

Comparing with sensitivity study

A sensitivity study is an evaluation of one or more processor parameters made by varying the parameters over a range of values, usually through many simulations. Interaction costs can be viewed as a way to *interpret* the data obtained from a sensitivity study. Regardless of how they are computed, through multiple simulations or graph analysis, interaction costs explain *why* performance phenomena occur in a very *concise* way.

Let's explore this relationship by validating that the conclusions obtained from interaction-cost analysis and conventional sensitivity studies are the same. We perform the comparison by using a corollary of the serial interaction between the instruction window and load latency (the main result of Section 7.3.1). As the load latency becomes larger, increasing the size of the instruction window has increasing benefit. Since load latencies and window stalls occur in series with each other (because EP edges are in series with CD edges, as can be seen in Figure 7.7), increasing the latency of one will make both more dominant on the critical path.

Using this corollary, we performed the comparison by running several simulations to observe the speedup with increasing window size at different cache latencies (see Figure 7.8). Indeed, the interaction costs correctly predicted what the sensitivity study reveals: for instance, 50% greater speedup ((9-6)/6 x 100%) is obtained from increasing the window size from 64 to 128 when the

data-cache latency is four instead of one.

From this example, we see the relationship between the two types of analyses. A full sensitivity study provides more information, *e.g.*, whether the curves in the plot are concave or convex; but interaction costs provide easier *interpretation* and concise *communication* of results. The interpretation is easy since the type and magnitude of the icosts have well defined meanings. The ease in communication comes from the ability to summarize a large quantity of data very succinctly. For example, the entire chart of Figure 7.8 can be summarized by simply stating that the two resources interact serially. Furthermore, due to the formulaic nature of interaction cost, the interpretation is available *automatically*, without the effort of a human analyst.

Summary. In this section, we showed that interaction costs can help microarchitects during the design process. When the dependence graph is constructed by the simulator, architects can use interaction-cost-based breakdowns as a standard output of each simulation run. The overhead of building the graph during simulation in our research prototype is approximately a two-fold slow-down, which we did not find overly burdensome, considering the substantial benefit of the added insight. Furthermore, using the same principles of sampling that facilitate the profiling solution of Section 6.3, we found that the overhead could be reduced to approximately 10% without significantly impacting accuracy (with only 1–2% error due to sampling).

7.3.2 Using Criticality in Design (Work by Others)

Before leaving the section on using criticality in hardware design, we discuss two works by others that fall into this category. The first used the a derivative of our model to gain insight very early in the design process. Specifically, they used criticality to understand the tradeoffs between

dataflow and superscalar processors. The second used criticality breakdowns to gain insight into the performance of their proposed new microarchitectural feature.

Comparing dataflow and superscalar processors. Budiu, *et al.* [17] use the critical path to understand the limitations of the traditional dataflow model compared to speculative out-of-order processors. They find the dependences eliminated by superscalar processors through speculation (control and data) are on the critical path of dataflow execution of many programs, accounting for much of the performance advantage of superscalar processors.

Simple Criticality Breakdowns. Petric and Roth [71] and Petric, *et al.* [72] used our graph model to compute critical-path breakdowns of the execution to better understand why the optimizations they proposed worked better on some benchmarks than others. They discovered, for instance, that their RENO optimizer did not effectively tackle memory bottlenecks and, when it was successful, their optimizer caused the machine to become more fetch bound (*i.e.*, the execution time was primarily determined by the instruction fetch engine). Thus, coupling RENO with increased fetch bandwidth could yield higher speedups.

Criticality Analysis of Clustered Processors. Salverda and Zilles [85] use a critical-path analysis similar to our criticality modes discussed in Section 4.1. This analysis helped them discover several important characteristics of the performance of machines with clustered execution units. For one, the criticality analysis showed them when the machine shifted from being fetch-critical to execute-critical due to the extra latency imposed by the clusters. It also showed what component of the steering policy was most responsible for the slowdown. (For their policy the largest contributer was load-balance steering, which causes an instruction to be sent to the least-filled cluster when its most

desired cluster is full.)

They also discovered that contention for functional units between predicted-critical instructions was a significant cause of the slowdown. This discovery led them to introduce a refined predictor based on the *likelihood of criticality*. With this metric, a static instruction is not only predicted critical or not but, instead, the criticality is weighted by the percentage of its previous dynamic instances that were critical. If 40% of a static instruction's dynamic instances are critical, it would get priority over one that was critical only 20% of the time. With these improvements, they were able to obtain performance on a clustered machine that is only a few percent worse than a monolithic one.

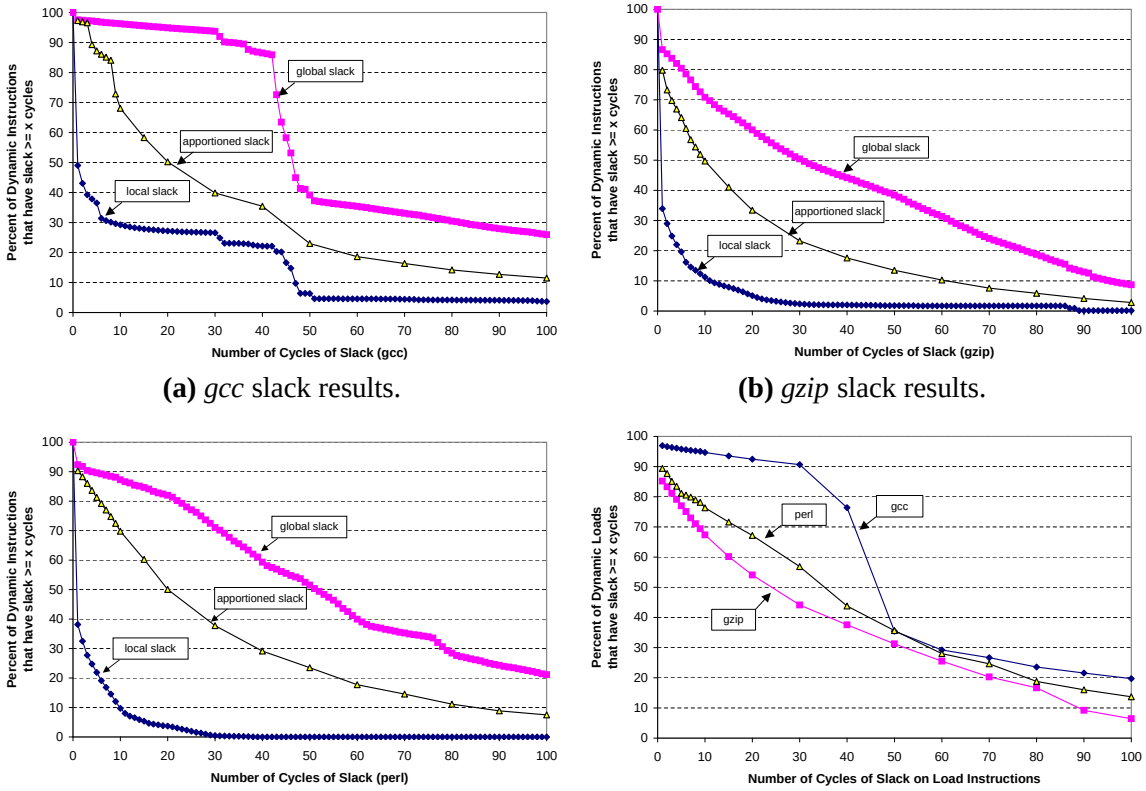
7.4 Software Design Help

Criticality can be useful to performance-conscious software engineers for a variety of purposes. The simplest is better understanding of what portions of code take the longest to execute, focusing optimization efforts. Since, in machines that exploit parallelism, performance counters are not sufficient for recognizing cost in this manner, cost and interaction costs could be a valuable addition to a profiling tool, such as Intel's Vtune [24].

While we believe there are great oppurtunities in improving software through criticality analysis, in particular using the shotgun profiler, our research has not delved much into those possibilities. In the next chapter, however, we do discuss how criticality analysis could be useful when writing multithreaded applications, which will be a very important problem with the increasing popularity of chip multiprocessors.

7.5 Summary

In this section we showed how alterations to our dependence graph can illustrate the performance effect of a software change without actually performing the change. This capacity enables software engineers to try out more configurations than otherwise would be feasible. Criticality analysis can also help in deciding where to place prefetch instructions, predicate branches, cut a program into threads, or — in general — where to focus human optimization effort. We need criticality for these tasks since increasing parallelism is making simple event counters less and less representative of what are the most important factors of execution time.



(a) *gcc* slack results. **(b)** *gzip* slack results.

(c) *perl* slack results. **(d)** All slack apportioned to loads.

Figure 7.2: **Across benchmarks, there is enormous potential for exploitation of slack. (a)-(c)** Measurements of local, apportioned, and global slack for SPEC2000 versions of *gcc*, *gzip*, and *perl*. *gcc* and *gzip* represent the two extremes in the amount of slack available in the full set of benchmarks we ran; *perl* is more typical. The measurements indicate that even in the least slackful benchmark, *gzip*, there is enormous potential for hiding delays introduced by nonuniform machines. **(d)** Measurements of apportioned slack when all available slack is apportioned to load instructions. These results show it may be possible to tolerate technologically-induced bottlenecks on load instructions if, for instance, wire delays cause some instructions to endure longer L1 data cache access times than others.

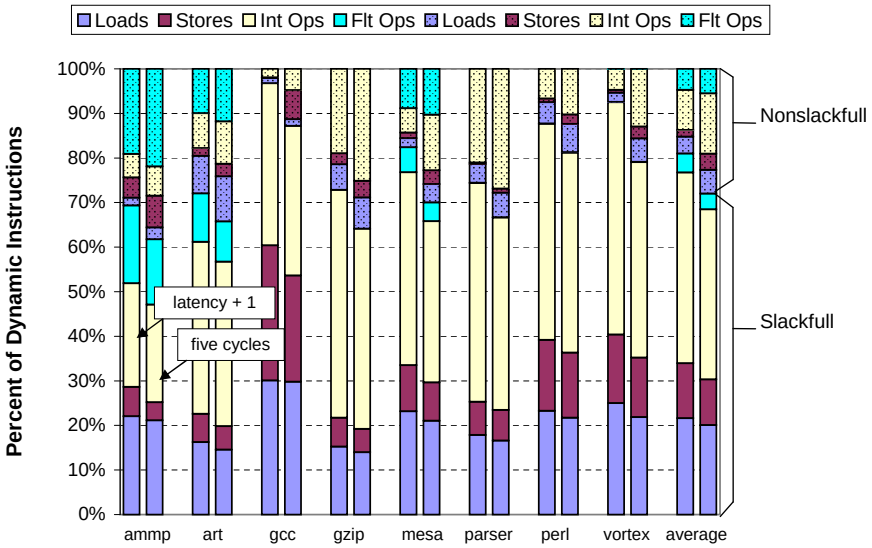


Figure 7.3: **Limit studies.** Measurements for two apportioning strategies are shown: *latency-plus-one-cycle* and *five-cycle* apportioning. These measurements provide an indication as to what types of non-uniform machine designs can be tolerated by a slack-based policy. For instance, latency-plus-one-cycle apportioning is relevant for the fast/slow pipeline microarchitecture we study in this thesis.

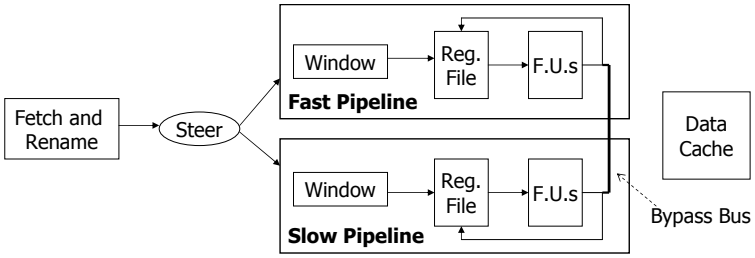


Figure 7.4: **The non-uniform microarchitecture used in our experiments.** The processor consists of one fast and one (or two) slow pipelines.

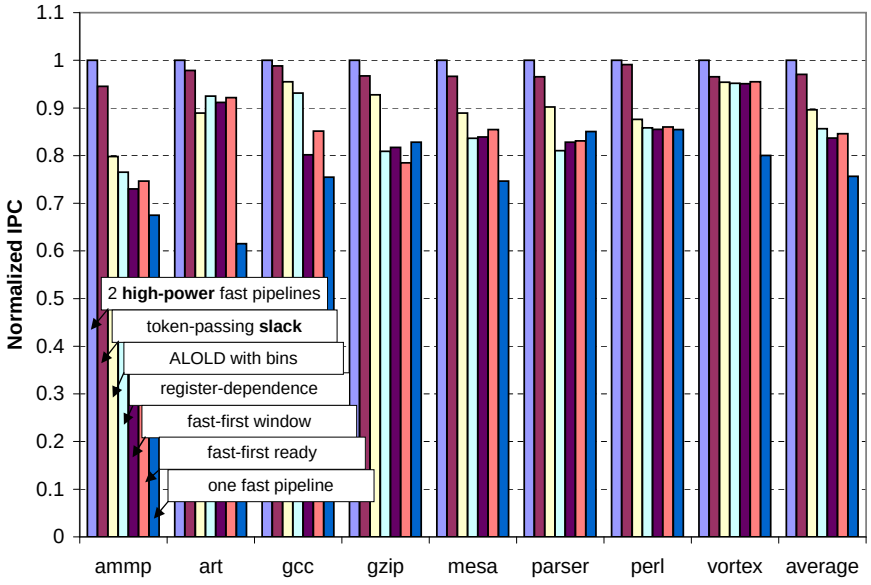
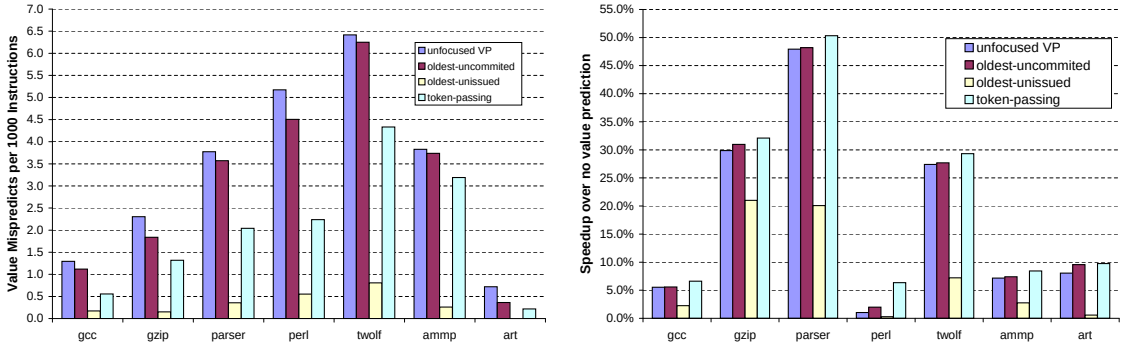


Figure 7.5: **Comparing control policies on fast/slow pipeline microarchitecture.** All measurements are normalized to the baseline of two fast 3-wide pipelines (*3f+3f*). Also, results are shown for a single fast 3-wide pipeline (*3f*) for reference. The rest of the measurements are different control policies for a *3f+3s* machine.



(a) Value misspeculations. **(b)** Speedup of focused value prediction.

Figure 7.6: **Focusing value-prediction by removing misspeculations on non-critical instructions. (a)** A critical-path predictor can significantly reduce misspeculations. **(b)** For most benchmarks, the token-passing critical-path predictor delivers at least 3-times more improvement than either of the heuristics-based predictors.

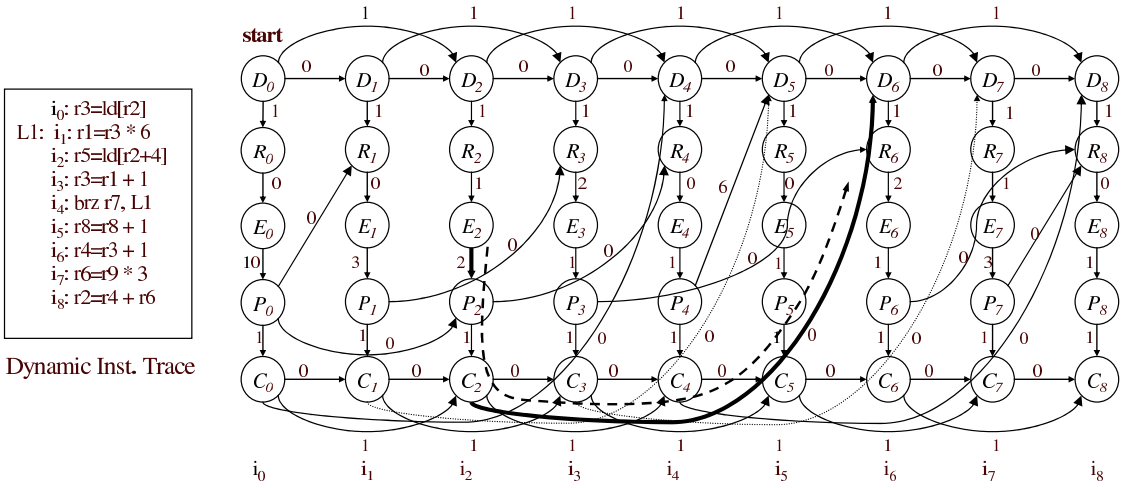


Figure 7.7: **Illustration of interaction between load latency and the instruction window** The dashed arrow shows how some load access *EP* edges and *CD* window edges are in series and, thus, have the potential to interact serially (see Section 7.3.1). Note that some other *EP* and *CD* edges are in parallel, thus there is also potential for parallel interaction between loads and the finite window constraint.

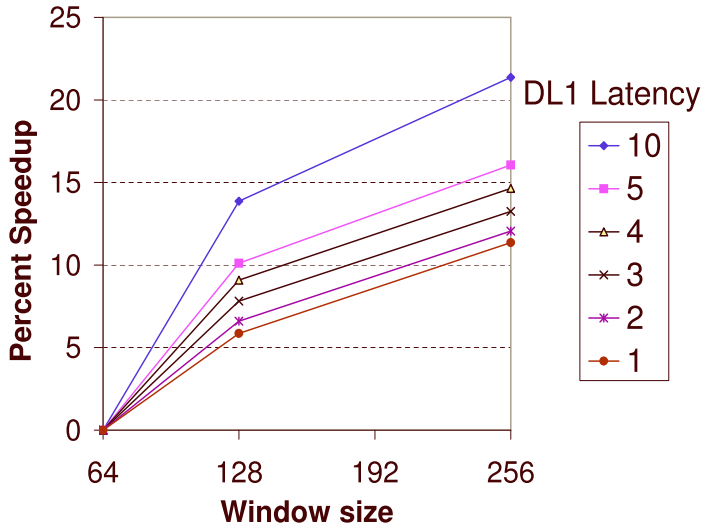


Figure 7.8: **Speedup from increasing window size for different level-one cache latencies.** As predicted from the negative interaction cost, increasing the window size has a larger benefit when level-one cache latencies are larger.

Chapter 8

Future Work: Criticality in Chip Multiprocessors

Our work has focused primarily on analyzing and exploiting the criticality characteristics of complex superscalar processors. There is a recent trend in microarchitecture design, however, towards multiple simpler cores that coordinate to provide increased performance for applications. In this chapter, I will outline some ways in which criticality can be used to improve the usabilty and effectiveness of these chip multiprocessors.

There has been some research already that has adapted our work to the newly popular domain. In particular, Li, *et al.* [61] extended our critical-path model to be applicable to parallel multithreaded applications. They showed how analyzing the critical path can provide insights into how a program is performing. For example, they can identify which threads are very costly in terms of execution time and which have slack and can be delayed.

The basic mechanism used in the work by Li, *et al.* was to first create a dependence

graph for each thread independently and then link the threads with dependence edges corresponding to synchronizing communication. The extra dependence edges connect execute (E) nodes of one application to another. Their methodology is most useful in analyzing existing multithreaded applications to help identify where software improvements could be made.

8.1 Software Parallelization

In the rest of this section, we outline a technique for using criticality to help with a different task, parallelizing a single-threaded application to run on multiprocessor hardware. This task will be at the forefront of the agenda for those software engineers that want their programs to take full advantage of chip multiprocessors. The complete task of automatically parallelizing a single-threaded application is a very challenging unsolved problem that is beyond the scope of our work. Nonetheless, we will show how the graph model and criticality analysis has promise for aiding software designers in the parallelization effort. Our hope is that the reasoning used in adapting our graph models and applying the criticality analysis will be useful in helping future researchers and designers in developing new techniques for chip multiprocessors.

Specifically, we will show new insights into simplified versions of the problem that may still be realistic enough to be useful in practice. The primary simplification we impose is that no changes to the binary are allowed. We also assume a multiscalar-like [100] execution model, where sequential tasks are extracted from the program stream and assigned to processors that communicate in a round-robin fashion (illustrated in Figure 8.1). For simplicity, we'll assume the communication is very efficient, (*i.e.*, through direct register-register transfers and a shared cache), but the models we describe could be altered to model longer latencies. Stated succinctly, the problem we will look

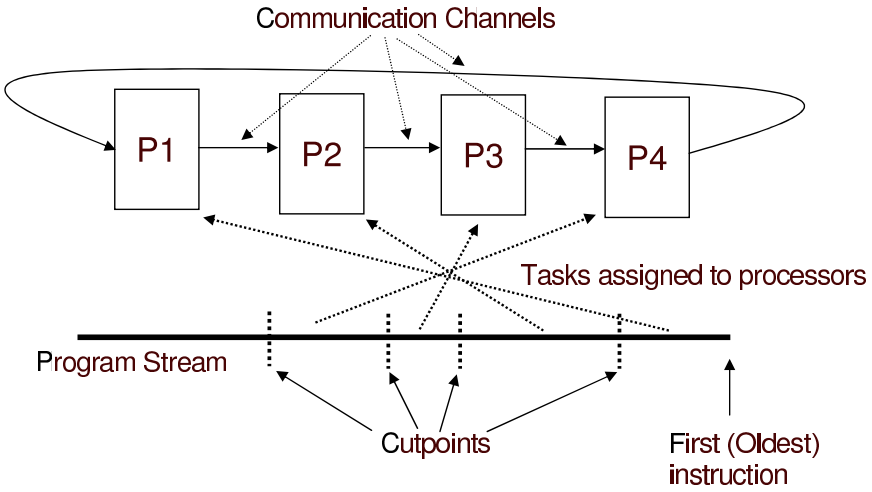


Figure 8.1: **Assumed Execution Model.** For the software parallelization case study, we assume a Multiscalar-like execution model like the one pictured above.

at is *Given an existing binary, what are the best "cut-points" for dividing it into* ***n*** *threads?*

There are many factors that must be considered to answer such a question. For example, the nature of *data dependences* in the program will determine the inter-thread communication. *Control dependences* are obviously important since they determine whether a segment of code is going to be executed at all. Achieving proper *load balancing* among processors represents a host of other challenges, such as determining the amount of work required to execute each thread — which is dependent upon the execution latency of instructions, data dependences, cache behavior and branch prediction among a myriad of other microarchitectural factors affecting performance.

One of the most useful aspects of our dependence graph abstraction of program performance is that we do not need to model these factors separately in order to devise an analysis that is faithful to all of them. Instead, the dependence graph provides most of this detail *uniformly* once an accurate model is developed. Next we'll discuss how it can be used for our present purpose.

8.1.1 Modeling multithreaded program execution

The first version of the problem we will attempt to address is a tool for software engineers. Assume that the programmer picks a section of code that he would like to split off as a separate thread, say a particular subroutine call. Effectively, creating this thread involves "cutting" the program before and after the subroutine call. Our goal will be to tell the programmer what performance improvement he should expect from the specified cut, without requiring any human effort in implementing synchronization and communication, nor any time-consuming recompilation. In fact, we would like to know the speedup very quickly so that the programmer can try out many alternatives until he finds one that works well.

We will use our graph model to provide such a tool. Since the models presented thus far have been for the execution of a program on a single superscalar, out-of-order core, we need to modify this graph so that it models the same program executing on multiple cores. Let's start with the simplest possible example. Assume the dynamic program execution is 1,000 instructions long and we want to break into two 500 instruction threads for execution on two processors. How could we use the graph to find the speedup from this optimization?

The most important property to model is that (in a multiple processor system) the two threads can be started at the same time. In other words, instructions i and $i + 500$ can be fetched at the same time, each on its own processor. In terms of microarchitectural constraints, there is no in-order fetch dependence between i and $i + 500$. The single-core model, however, includes such a dependence (transitively) with the FF edges from instruction i to $i + 1$ to $i + 2$ all the way to $i + 500$. To model the multithreaded execution, we need to relax this constraint.

The simplest way to relax the in-order fetch constraint in this instance is to remove the FF

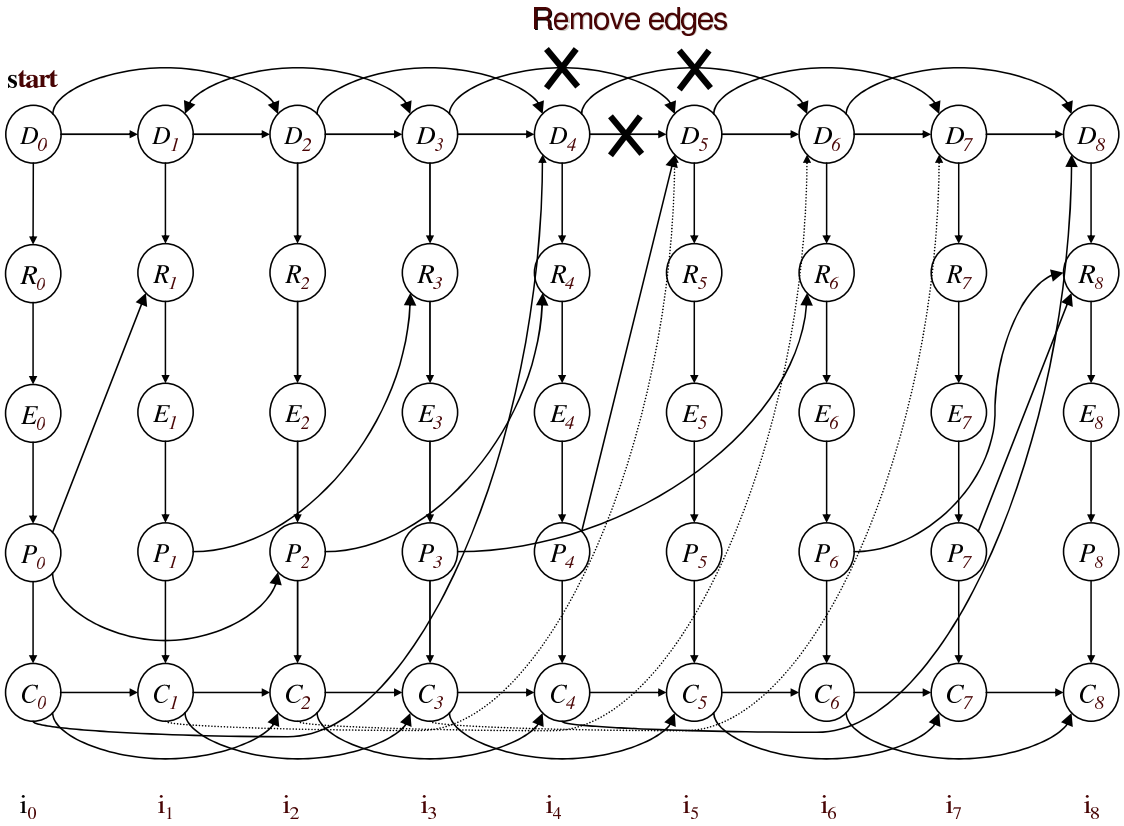


Figure 8.2: **Cutpoint illustration.** If the eight instruction program represented by the graph above were to be cut into two threads, one thread consisting of instructions 1,2,3,4, and 5 and a second thread consisting of instructions 6, 7, and 8, the execution-time improvement could be measured by removing the edges marked with an "**X**" and observing the resulting decrease in critical-path length.

edge from instruction $i + 499$ to $i + 500$. This way, the fetch node of $i + 500$ is "fired" immediately, at the same time as the fetch node for instruction i. An illustration of this simple graph manipulation is shown in Figure 8.2.

Notice how the model naturally accounts for communication between the threads: the data dependence (EE) edges remain intact, so that the later thread will have to stall if the earlier thread has yet to produce data that it needs. In a real system, some extra latency would result due to this communication. We can enhance the model to account for that by increasing the weights (*a.k.a.*, latencies) on the appropriate EE edges. If other chip-multiprocessor specific constraints are

Test the benefit of cutting the program at static PC x

1. Construct graph of the dynamic program execution on a single core.
2. Remove the FF edges immediately preceeding each instance of x.
3. Measure the critical path length. This is an estimate of the execution time of the parallelized program.

deemed important, we could continue to tune the model to be as detailed as needed. One of the nice features of the model is that approximate results can be collected very easily and rapidly and then made more precise over time as effort is expended tuning the latencies and constraints.

Using the model, it is clear how to test a programmer-specified cutpoint. For instance, splitting off a procedure call as another thread involves making cuts before and after each dynamic instance of that call. Thus, the first step is constructing a graph of the program executing on a single core, which could be obtained either through shotgun profiling or simulation. Then the graph modification described above, removing the FF edge, would need to be performed for each cut. The resulting critical-path length would be an estimation of the runtime of the new multithreaded program. Since finding the critical-path length can be done rather quickly, the programmer would be able to test out many different parallelized variations of his program. The algorithm is shown at a high-level in Figure 8.1.1.

8.1.2 Automatic Parallelization

Above we presented a tool to help software engineers decide whether a parallelization that they propose via intuition is indeed good for performance or not. A more challenging problem is to come up with the thread cutpoints automatically, without human intuition. We cannot solve that more general problem, but we will discuss in this section how interaction costs may be useful in such an effort.

Let's say we have a relatively short segment of code that is to be divided into N threads for execution on N processors. The goal here is to find the $N-1$ cutpoints in this segment that would yield the highest performance possible. Stated in terms of our $cost$ metric, the goal is to maximize $cost(\{p_1, p_2, .., p_{N-1}\})$, where p_1 to p_{N-1} are the FF edges to be cut in the graph.

Unfortunately, finding a set of optimal cutpoints is an NP-complete problem. We can use interaction cost, however, to help us converge on a good, if not optimal, solution quickly. To see how, consider the interaction cost of two cutpoints p_1 and p_2. If $icost(\{p_1, p_2\}) << 0$, indicating a serial interaction between the two cutpoints, we know that some of the benefit obtained from p_1 is also obtained by p_2. In other words, the two cutpoints are doing redundant work, eliminating many of the same execution cycles. This behavior is probably a result of p_1 and p_2 being placed too close to each other in the program stream. Another way to look at it is the thread between p_1 and p_2 is too small relative to the other threads. A good heuristic could be designed to take advantage of this effect, looking for serial interactions between adjacent cutpoints and moving them further apart if the magnitude is too high.

Figure 8.3(a) shows some results from preliminary experiments measuring the distribution of costs for every possible dynamic cutpoint. In other words, the benefit, in terms of execution time reduction, was measured for cutting the program into two threads between every pair of consecutive instructions in the program. If we did not have the graph, many thousand simulations would need to be run for each benchmark to obtain the same results.

The results indicate that relatively few cutpoints yield substantial execution time savings. In fact, for most benchmarks, greater than 70% of the possible cutpoints yield benefits of less than 10 cycles. This suggests that a cost-sensitive policy for choosing cutpoints may be important for

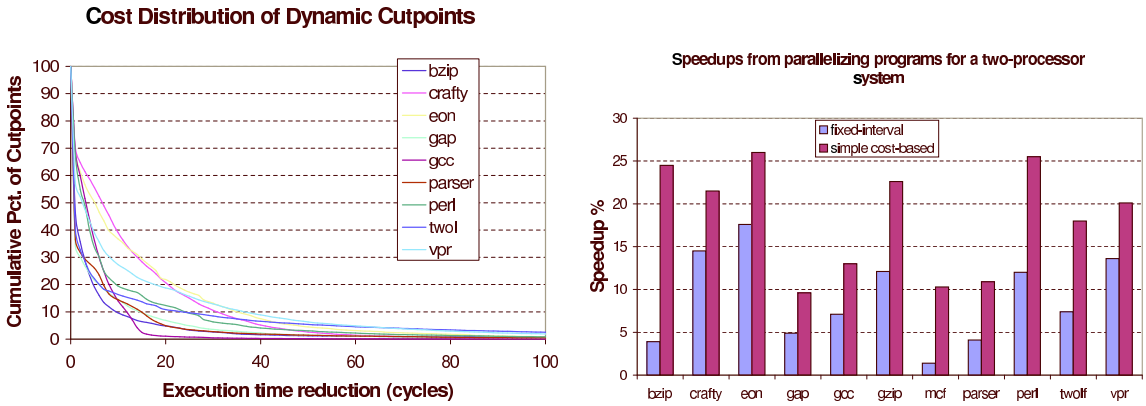


Figure 8.3: **Distribution of execution-time reduction from cutpoints.** The cumulative distribution shown in the charts is the mirror image of how they are often displayed. In other words, from (a), for all benchmarks, greater than 75% of the dynamic cutpoints improve performance by less than 20 cycles. **(a)** is the cost distribution observed from cutting a program into two threads between each pair of consecutive *dynamic* instructions. **(b)** Speedup from parallelizing a program for a machine with two processors. The *fixed-interval* policy creates a cutpoint every 100 dynamic instructions. The *simple cost-based* policy picks as a cutpoint the dynamic instruction with the highest singleton cost (ignoring interactions) in every 100 instruction interval. The purpose of this experiment is to show that cost-sensitive policies for parallelizing applications can be beneficial. Due to the simplicity of the policy, however, it does not provide much insight into the best achievable speedup.

achieving the best performance.

As a crude test of this hypothesis, Figure 8.3(b) shows the speedup obtained from parallelizing the benchmarks using two different policies, one where a cut in the program was made every 100 dynamic instructions and another where a cut was made at the highest singleton cost within each 100 dynamic-instruction interval. Thus, both policies create the same number of threads for each benchmark. The cost-sensitive policy achieved 1.5–7 times the speedup of the fixed-interval policy.

8.2 Summary

Although the bulk of our work has focused on complex superscalar processors, our criticality techniques are not limited to only this style of processor implementation. In this chapter, we illustrated how our dependence graph and criticality analysis could be adapted for one architectural style that is gaining in popularity: chip multiprocessors. There is a lot of work to do beyond what is presented here to fully adapt our technology, but we hope that this chapter gave a reasonable introduction as to how that work might proceed.

Chapter 9

Conclusions and Future Work

Our work set out to deal with the inadequacy of event counts for judging performance, both in optimizations and pure analysis. We quickly realized that the key ingredient to a performance analysis methodology for parallel microarchitectures is an understanding of the critical path through a program execution. Once you have the fundamental understanding of what the critical path is and how to measure it, a remarkably large class of performance analysis questions can be answered.

Of course, the critical path has long been used by compiler writers and others as an aid in making optimization decisions. In these cases, the critical path was found on a graph consisting of instructions interconnected by data dependences. This paradigm of thinking of nodes as instructions and edges as data dependences inhibited architects from effectively exploiting the critical path through microprocessor program executions. Intuitively, it was clear that there existed limiters to performance other than data dependences, but it wasn't clear how to deduce a global critical path that included them. Our insight of breaking a program's execution into smaller bits than just instructions enabled a more complete modeling of performance.

In this thesis, we attempted to lay the groundwork for a successful performance analysis. Nonetheless, there are still many topics that merit further exploration. Below we discuss a few potential research projects.

Modeling and Queuing Theory. Certain types of hardware resources for which an instruction could use any one of many, such as functional units, cause modeling difficulties using our techniques. We discussed some possible work-arounds for this problem in Chapter 3, but a complete solution probably requires incorporating new ideas from another domain, perhaps queuing theory. Our preliminary work in this area revealed that it may be a very difficult task, since a standard memoryless queue did not provide any increased accuracy.

Automatic Model Deduction. In this thesis, all of the graph models have been constructed by hand, using our human intuition. Since this procedure requires not only knowledge of the processor but also understanding of how to use a graph to model various features, a better solution would be to deduce the model automatically from a specification. This specification should be of a standard format that designers are naturally accustomed to using during their normal design effort. In fact, some work has attempted to deduce a model directly from the RTL specification [16]. While this may be useful, the models constructed from RTL may be too dense for many practical purposes. A higher-level specification language that could be used even in early stages of the research and design process would ease the use of our criticality analysis.

Criticality in Other Domains. Our basic graphical analysis and metrics, including the characterization of interactions, is applicable to any parallel system, not just complex microarchitectures. We discussed some preliminary work on applying these techniques to chip multiprocessors in Chapter 7.

Previous work, discussed in Chapter 2, used the critical path to profile performance at a higher-level of abstraction, useful to programmers [61]. Many other domains might also benefit from this form of analysis as well. For example, network protocols might be optimized by measuring interaction costs of communication during typical patterns of use.

Application-Specific Analyzers. The work by Sasanka, *et al.* [87] used a token-passing hardware structure similar but simpler than our token-passing criticality analyzer for the very specific task of resizing the instruction window. The advantage of this application-specific approach is not just a simpler implementation, however. Sasanka, *et al.*'s work actually computes a reasonable approximation of the *cost* (as opposed to just criticality) of the instruction window. As we discussed in Chapter 6, building a general hardware *cost* estimator seems intractable, but it may be possible to do so on a limited basis for specific applications. For example, it would be very useful to have a hardware mechanism that could estimate the *cost* and interactions of cache misses (or branch mispredicts.)

Predictors for Fetch and Commit Criticality. The token-passing analyzer was meant primarily to detect the criticality of execute (or E) nodes, as opposed to fetch (F), commit (C), or nodes representing other micro-operations. There are many possible optimizations that target these other stages of instruction processor, however, and criticality could help guide them. For example, the limited space available in a trace cache could be more effectively utilized by storing critical fetch blocks. In fact, a frontend mechanism more effective than a pure trace cache may be possible if we can identify those fetch blocks that must be fetched quickly versus those that can be delayed.

In principle, the token-passing analyzer that works for E nodes also works for any of the

other nodes — all that is necessary is to plant a token into whatever node that should be tested for criticality. The analyzer is just as effective at detecting whether an F or C node is critical as it is for E nodes. The problem arises when attempting to predict criticality of future nodes based on past detections. We have observed that F and C criticality do not experience the same type of static instruction "locality" as do the E nodes. In other words, the typical approach of training a predictor that is indexed by PC does not work well.

A solution will likely have to rely on the characteristics of program executions peculiar to the specific micro-operations that are targeted. For example, we have noticed that fetch nodes are most often critical after disruptions in the program stream, such as those caused by branch mispredictions and, to a lesser extent, fetch stalls due to a full instruction window. Furthermore, we have found that the number of consecutive critical fetch nodes after a branch misprediction does have static instruction locality. In other words, when a dynamic instance of a particular static branch instruction is mispredicted, the length of the resulting critical chain of F nodes is fairly similar in length to that resulting from other mispredicted dynamic instances of the same static instruction. Recording this length could form the basis of an effective fetch node criticality predictor.

Using Interaction Costs. While we provided some heuristics and illustrated how to interpret and use interaction costs in this thesis, we still left much to be explored. In particular, we have found it challenging in practice to pick the correct interactions to measure. It's also difficult to reason intuitively about interactions between more than two events. This space provides ample room for new graph and, perhaps, data mining algorithms to extract useful information automatically.

Criticality in Configurable Hardware. In this thesis, we have assumed the traditional restrictions as to what is feasible to be done in hardware and these restrictions have formed the basis for our hardware/software boundaries. In particular, much care was taken to reduce the hardware requirements for the shotgun profiler. These simplifications sacrificed some amount of accuracy in the outcome of the criticality analysis in exchange for a feasible implementation.

Configurable hardware, such as the FPGAs common in ASIC development, present a different environment for deciding the hardware/software boundary. The fluidity of the hardware allows more or fewer transistors to be dedicated to analysis depending upon what is desired at different phases of ASIC development. For instance, while the most time-consuming portions of the application are being optimized, much of the chip real estate could be devoted to providing very accurate analysis. The most natural way to use the extra transistors would be to construct very accurate graphs by monitoring all (or most) of the dynamic operations as they occur. If it is not possible to record statistics for each dynamic instruction in a long stream, the extra transistors could still be used to increase the accuracy of the shotgun profiler by increasing the number of samples recorded.

A second way those extra transistors could be used is to compute the desired metrics in the hardware itself. For example, the effectively linear time algorithm used to compute the cost of each individual edge in a graph (see Section 5.3) could be implemented directly in the hardware. The advantage of this approach is that less information would need to be communicated to some external source for analysis, assuming that the result of the analysis is more compact than the information required to construct the graph. The hardware/software tradeoffs in this environment could be an interesting research topic.

Using Criticality in FPGA emulators. The advantages of the flexibility of configurable hardware in producing accurate analysis can be extended even to nonconfigurable processors by employing a system such as in the RAMP project [6]. This approach uses configurable hardware (*e.g.,* FPGAs) to model the functional and performance characteristics of nonconfigurable processors. Although the timing characteristics of FPGAs differs from custom hardware, emulation or scaling of latencies can be used to mimic the behavior of a full custom chip.

Our dependence graph and criticality analysis work very well with such a system. For the simplest example, the critical path through our dependence graph is equal to the execution time of the program. Thus, the dependence graph, along with the last-arriving rule, tells us exactly what the hardware needs to keep track of in order to report execution time accurately.

Perhaps more interestingly, however, the dependence graph provides a framework for how to report performance information back to the user of the system. In a typical software simulator, many (often redundant) performance counters are used to gain a sense of not only how well hardware performed but also why it performed the way it did. The dependence graph makes it easier to determine the minimal amount of information that must be extracted from an emulator in order to provide the user of the system a complete picture of performance. That dependence graph can be constructed using a technique such as the shotgun profiler, whose accuracy is tunable by increasing the number of samples that are captured.

Summary. In this dissertation, we have a provided a framework and tools to model, measure and interpret the criticality ($cost$, $slack$, and interaction) characteristics of program executions. We have also developed hardware support for detecting and predicting critical-path instructions and their slack for use in online optimizations. Finally, we have proposed a profiler to replace or

enhance traditional hardware performance counters — enabling sophisticated criticality analysis of real programs. There are many ways that the framework and hardware structures could be improved further, but perhaps the most exciting prospect is to use the existing tools for optimizations and analysis applications beyond the case studies explored in our work. As of the writing of this thesis, some researchers have already begun to do so, but there is much more to be explored.

Bibliography

[1] *Sixth International Symposium on High-Performance Computer Architecture*, Toulouse, France, Jan.

[2] V. Agarwal, M.S. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture (ISCA'00)*, Vancouver, June 10–14 2000.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Twenty-fifth International Conference on Very Large Databases, Edinburgh, Scotland, UK, 7–10 September, 1999*, pages 266–277, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers.

[4] A. R. Alameldeen, C. J. Mauer, M. Xu, M.M.K. Martin P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Computer Architecture Evaluation using Commercial Workloads (CAECW '02) in conjunction with HPCA '02*, February 2002.

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L.

Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, Nov 1997.

[6] Arvind, K. Asanovic, D. Chiou, J.C. Hoe, C. Kozyrakis, S.L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. Technical report, 2006.

[7] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *The* 19^{th} *International Symposium on Computer Architecture (ISCA)*, pages 342–351, 1992.

[8] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *28th International Symposium on Computer Architecture*, pages 218–229, Göteborg, Sweden, June30–July4, 2001. IEEE Computer Society and ACM SIGARCH.

[9] T. Ball and J. R. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, pages 46–57, 1996.

[10] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled superscalar processors. In 33^{th} *International Symposium on Microarchitecture*, Dec 2000.

[11] P. Barford and M. Crovella. Critical path analysis of tcp transactions. In *Proceedings of ACM SIGCOMM 2000*, January 2000.

[12] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory system characterization of commercial workloads. In *25th International Symposium on Computer Architecture (ISCA-98)*, pages 3–14, New York, June 27–July 1 1998.

[13] E. Borch, E. Tune, B. Manne, and J. Emer. Loose loops sink chips. In 8^{th} *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[14] E. L. Boyd and E. S. Davidson. Hierarchical performance modeling with MACS: A case study of the Convex C-240. In 20^{th} *International Symposium on Computer Architecture*, May 1993.

[15] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *34th International Symposium on Microarchitecture*, pages 204–213, Austin, Texas, December1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[16] M. Budiu, P. V. Artigas, and S. C. Goldstein. Dataflow: A complement to superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[17] Mihai Budiu, Pedro Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–186, Austin, TX, March 2005.

[18] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, Jun 1997.

[19] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In 26^{th} *International Symposium on Computer Architecture*, May 1999.

[20] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *Kool Chips Workshop in conjunction with MICRO 33*, Dec 2000.

[21] Y. Chin, J. Sheu, and D. Brooks. Evaluating techniques for exploiting instruction slack. In *International Conference on Computer Design*, October 2004.

[22] Y. Chou and J. P. Shen. Instruction path coprocessors. In *27th International Symposium on Computer Architecture*, pages 270–281, Vancouver, British Columbia, June12–14, 2000. IEEE Computer Society and ACM SIGARCH.

[23] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*, pages 14–25, Göteborg, Sweden, June30–July4, 2001.

[24] Intel Corporation. Vtune: A visual tuning environment. *http://support.intel.com/support/performancetools/vtune/*.

[25] Intel Corporation. Intel Itanium 2 processor reference manual for software development and optimization. Apr 2003.

[26] Intel Corporation. Intel Pentium 4 processor manual. In *[http://developer.intel.com/design/pentium4/manuals/]*, 2003.

[27] D. Crowe, G. A. Muthler, S. J. Patel, and Steven S. Lumetta. Instruction fetch deferral using static slack, January 30 2002.

[28] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In 30^{th} *International Symposium on Microarchitecture*, Dec 1997.

[29] A. El-Moursy, R. Garg, D.H. Albonesi, and S. Dwarkadas. Partitioning multi-threaded pro-

cessors with a large number of threads. In *International Symposium on Performance Analysis of Systems and Software*, March 2005.

[30] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In 29^{th} *Annual International Symposium on Computer Architecture*, pages 37–46, 2002.

[31] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In 34^{th} *International Symposium on Microarchitecture*, Dec 2001.

[32] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In 30^{th} *International Symposium on Microarchitecture*, pages 149–159, Los Alamitos, December 1–3 1997.

[33] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In 29^{th} *International Symposium on Computer Architecture*, May 2002.

[34] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In 28^{th} *International Symposium on Computer Architecture*, Jun 2001.

[35] B. A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In 36^{th} *International Symposium on Microarchitecture*, Dec 2003.

[36] B. A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Compiler Optimization*, 2004.

[37] B. A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Interaction cost: For

when event counts just don't add up. *IEEE Micro Special Issue: Micro's Top Picks from Microarchitecture Conferences*, 2004.

[38] Brian A. Fields. *Using Criticality To Attack Performance Bottlenecks*. PhD thesis, University of California—Berkeley, December 2006.

[39] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. Oct 1999.

[40] R. D. Fleischmann et al. Whole-genome random sequencing and assembly of haemophilus-influenzae. *Science*, 269:496–512, 1995.

[41] D. Folegnani and Antonio González. Energy-efficient issue logic. In 28^{th} *International Symposium on Computer Architecture*, July.

[42] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 272–281, New York, June 27–July 1 1998. ACM Press.

[43] S. Ghiasi, J. Casmira, and D. Grunwald. Using ipc variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design in conjunction with ISCA 2000*, Vancouver, British Columbia, June 2000. IEEE Computer Society and ACM SIGARCH.

[44] J. González and A. González. The potential of data value speculation to boost ILP. In *International Conference on Supercomputing*, pages 21–28, Melbourne, Australia, July13–17, 1998. ACM SIGARCH.

[45] J. González and A. González. Control-flow speculation through value prediction for superscalar processors. In *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 57–65, Newport Beach, California, October12–16, 1999. IEEE Computer Society Press.

[46] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10:9–15, October 1996.

[47] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In 29^{th} *International Symposium on Computer Architecture*, 2002.

[48] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 3^{rd} edition, 2002.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990.

[50] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[51] J. K. Hollingsworth and B. P. Miller. Parallel program performance metrics: A comparison and validation. In *Proc. Supercomputing*, Nov 1992.

[52] M. Horowitz and K. Mai R. Ho. The future of wires. In *Semiconductor Research Corporation Workshop on Interconnects for System on a Chip*, May 1999.

[53] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar.

The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In 29^{th} *International Symposium on Computer Architecture*, 2002.

[54] Raj Jain. *The Art of Cumpter Systems Performance Analysis*. Wiley Professional Computing, 1991.

[55] M. Johnson and K. Roy. Optimal selection of supply voltages and level conversions during data path scheduling under resource constraints. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 72–77, Washington - Brussels - Tokyo, October 1996. IEEE Computer Society.

[56] I. Kadayif and M. T. Kandemir. An integer linear programming based approach for parallelizing applications in on-chip multiprocessors. In *Design Automation Conference*, Jun 2002.

[57] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In 31^{st} *International Symposium on Computer Architecture*, Jun 2004.

[58] G. Kemp and M. Franklin. Pews: A decentralized dynamic scheduling algorithm for ilp processing. In *International Conference on Parallel Processing*, pages 239–246, Aug 1996.

[59] D. R. Kerns and S. J. Eggers. Balanced scheduling: instruction scheduling when memory latency is uncertain. *ACM SIGPLAN Notices*, 28(6):278–289, June 1993.

[60] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *The* 19^{th} *International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.

[61] T. Li, A.R. Lebeck, and D. J. Sorin. Quantify instruction criticality for shared memory

multiprocessors. In 25^{th} *Symposium on Parallelism in Algorithms and Architectures*, June 2003.

[62] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, pages 226–237, Paris, France, December2–4, 1996.

[63] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In 29^{th} *International Symposium on Microarchitecture*, Dec 1996.

[64] D. Marculescu. Application adaptive energy efficient clustered architectures. In *International Symposium on Low Power Electronics and Design*, August 2004.

[65] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading techonology architecture and microarchitecture. *Intel Technology Journal*, 6:4–15, February 2002.

[66] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Trans. Parallel and Distributed Syst.*, 1(2):206–217, 1990.

[67] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *32nd International Symposium on Microarchitecture*, pages 147–155, Haifa, Israel, November16–18, 1999.

[68] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In 3^{rd} *International Symposium on High Performance Computer Architecture*, Feb 1997.

[69] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th International Symposium on Computer Architecture*, pages 206–218, 1997.

[70] S. Patel, M. Evers, and Y. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In 25^{th} *International Symposium on Computer Architecture*, Jun 1998.

[71] V. Petric and A. Roth. Energy aspects of pre-execution and energy-aware p-thread selection. In 32^{th} *International Symposium on Computer Architecture*, June 2005.

[72] V. Petric, T. Sha, and A. Roth. Reno: A rename-based instruction optimizer. In 32^{th} *International Symposium on Computer Architecture*, June 2005.

[73] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *Workshop on Complexity-Effective Design in conjunction with ISCA 2001*, Goteborg, Sweden, June 2001. IEEE Computer Society and ACM SIGARCH.

[74] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In 34^{th} *International Symposium on Microarchitecture*, December 2001.

[75] R. Rakvic, B. Black, D. Limaye, and J. P. Shen. Non-vital loads. In 8^{th} *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[76] R. Rakvic, B. Black, D. Limaye, and J.P. Shen. Non-vital loads. In 8^{th} *International Symposium on High-Performance Computer Architecture*, February 2002.

[77] Ryan Rakvic, Deepak Limaye, and John P. Shen. Non-vital loads. Technical Report CMuART-2000-02, Carnegie Mellon University, 2000.

[78] P. Ramarao. An adiabatic framework for a low energy u-architecture and compiler. In *Workshop on Interaction Between Compilers and Computer Architecture*, July 2003.

[79] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. Oct 1998.

[80] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In 15^{th} *Symposium on Operating Systems Principles*, Dec 1995.

[81] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture*, pages 24–34, Paris, France, December2–4, 1996.

[82] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *30th IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 138–148, Los Alamitos, December 1–3 1997.

[83] A. Roth and G.S. Sohi. Speculative data-driven sequencing for imperative programs. Technical Report CS-TR-2000-1411, University of Wisconsin, Madison, February 2000.

[84] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In 7^{th} *International Symposium on High-Performance Computer Architecture*, Jan 2001.

[85] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In 38^{th} *International Conference on Microarchitecture*, Dec 2005.

[86] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A design space evaluation

of grid processor architectures. In *34th IEEE/ACM International Symposium on Microarchitecture (MICRO-01)*, Austin, TX, December 2001.

[87] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[88] T. Sato and I. Arita. Energy reduction via critical path prediction. In *Workshop on Complexity-Effective Design*, May 2002.

[89] M. Schlansker and V. Kathail. Acceleration of algebraic recurrences on processors with instruction level parallelism. technical report HPL-93-55, HP Laboratories, 1993.

[90] M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 406–429, Portland, Oregon, August12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[91] M. Schlansker and V. Kathail. Techniques for critical path reduction of scalar programs. *International Journal of Parallel Programming*, 25(3):147–181, June 1997.

[92] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *27th International Symposium on Microarchitecture*, pages 40–51, San Jose, California, November30–December2, 1994.

[93] M. Schlansker, V. Kathail, and S. Anik. Parallelization of control recurrences for ILP processors. *International Journal of Parallel Programming*, 24(1):65–102, February 1996.

[94] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 155–168, 1999.

[95] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In 8^{th} *International Symposium on High-Performance Computer Architecture*, Feb 2002.

[96] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In 34^{th} *International Symposium on Microarchitecture*, Dec 2001.

[97] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In 34^{th} *International Symposium on Microarchitecture*, Dec 2001.

[98] J.E. Smith. Instruction level distributed processing. In 7^{th} *International Conference on High Performance Computing*, Dec 2000.

[99] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In 24^{th} *International Symposium on Computer Architecture*, 1997.

[100] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd International Symposium on Computer Architecture (22nd ISCA'95)*, pages 414–425, Santa Margherita, Italy, June 1995. Published as Proc. of the 22nd International Symposium on Computer Architecture (22nd ISCA'95) ACM SIGARCH Computer Architecture News, volume 23, number 6.

[101] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In 29^{th} *International Symposium on Computer Architecture*, 2002.

[102] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, Jul 2002.

[103] S. T. Srinivasan, R. Dz ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In 28^{th} *International Symposium on Computer Architecture*, Jun 2001.

[104] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In 31^{st} *International Symposium on Microarchitecture*, Nov 1998.

[105] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2004.

[106] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *30th IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 34–45, Los Alamitos, December 1–3 1997.

[107] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In 27^{th} *International Symposium on Computer Architecture*, Jun 2000.

[108] N. Tuck and D. M. Tullsen. Multithreaded value prediction. In 11^{th} *International Symposium on High-Performance Computer Architecture*, February 2005.

[109] D. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, Oct 1998.

[110] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In 7^{th} *International Symposium on High-Performance Computer Architecture*, Jan 2001.

[111] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In 11^{th} *International Conference on Parallel Architectures and Compilation Techniques*, Sep 2002.

[112] Kimiyoshi Usami, Mutsunori Igarashi, Takashi Ishikawa, Masahiro Kanazawa, Masafumi Takahashi, Mototsugu Hamada, Hideho Arakida, Toshihiro Terazawa, and Tadahiro Kuroda. Design methodology of ultra low-power MPEG4 codec core exploiting voltage scaling techniques. In *1998 Conference on Design Automation (DAC-98)*, pages 483–488, Los Alamitos, CA, June15–19 1998. ACM/IEEE.

[113] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 281–291, Los Alamitos, December 1–3 1997.

[114] J. Weber and E. Myers. Human whole genome shotgun sequencing. In *Genome Research*, pages 401–409, 1997.

[115] Jerome D. Wiest and Ferdinand K. Levy. *A Management Guide to PERT/CPM*. Prentice-Hall, 1974.

[116] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look into the future. In *Cool-Chips Tutorial in cunjunction with MICRO 1999.*, Nov 1999.

[117] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In 9^{th} *International Symposium on High Performance Computer Architecture*, Feb 2003.

[118] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculative techniques for improving load related instruction scheduling. In *26th International Symposium on Computer Architecture*, page ???, June 1999.

[119] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing '96*, 1996.

[120] W Zhang, N Vijaykrishnan, M Kandemir, M.J. Irwin, D. Duarte, and Y-T. Fai. Exploiting vliw schedule slacks for dynamic and leakage energy reduction. In *To Appear in 34th International Symposium on Microarchitecture*, Austin, Texas, December 2001. IEEE Computer Society and ACM SIGARCH.

[121] Q. Zhao and D.J. Lilja. Using compiler-generated approximate critical path information to prioritise instructions for value predictions. *Computers and Digital Techniques*, 151(5), 2004.

[122] C. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th International Symposium on Computer Architecture (ISCA'00)*, Vancouver, June 10–14 2000.

[123] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th International Symposium on Computer Architecture*, pages 172–181, Vancouver, British Columbia, June12–14, 2000.

[124] C.B. Zilles and G.S. Sohi. A Programmable Co-processor for Profiling. Jan 2001. Proc. 7th International Symposium on High Performance Computer Architecture.

[125] C.B. Zilles and G.S. Sohi. Execution-based Prediction Using Speculative Slices. July 2001. Proc. 28th International Symposium on Computer Architecture.