Slack: Maximizing Performance Under Technological Constraints

Brian Fields Rastislav Bodík Mark D. Hill

Computer Sciences Department
University of Wisconsin–Madison
{fields,bodik,markhill}@cs.wisc.edu

Abstract

Many emerging processor microarchitectures seek to manage technological constraints (e.g., wire delay, power, and circuit complexity) by resorting to *non-uniform* designs that provide resources at multiple quality levels (e.g., fast/slow bypass paths, multi-speed functional units, and grid architectures). In such designs, the constraint problem becomes a control problem, and the challenge becomes designing a *control policy* that mitigates the performance penalty of the non-uniformity. Given the increasing importance of non-uniform control policies, we believe it is appropriate to examine them in their own right.

To this end, we develop *slack* for use in creating control policies that match program execution behavior to machine design. Intuitively, the slack of a dynamic instruction i is the number of cycles i can be delayed with *no effect on execution time*. This property makes slack a natural candidate for hiding non-uniform latencies.

We make three contributions in our exploration of slack. First, we formally define slack, distinguish three variants (*local*, *global* and *apportioned*), and perform a limit study to show that slack is prevalent in our SPEC2000 workload. Second, we show how to predict slack in hardware. Third, we illustrate how to create a control policy based on slack for steering instructions among fast (high power) and slow (lower power) pipelines.

1 Introduction

Recent years have witnessed a proliferation of technology *constraint-aware* design proposals. For example, physical clustering of functional units has attacked wire delays [7, 8], multi-frequency functional-units have addressed power consumption [13], and grid architectures have sought to reduce cycle time [11]. More importantly, it appears that wire, power, and circuit-complexity trends will make constraint-aware designs even more prevalent in the future.

A challenging feature of many constraint-aware designs is that they introduce *non-uniformity*, where one or more resources are available at multiple "quality" levels. For example, clustering introduce bypasses of multiple latencies, multi-speed functional units offer several execution latencies and effective issue bandwidths, and grid architectures come with a non-uniform L1-cache latency.

A key observation thus is that constraint-aware designs often turn the *constraint problem* into a *control problem*. Using a *control policy*, these designs hide non-uniformity by steering each dynamic instruction to an appropriate resource. For example, clustering comes with a register-dependence instruction-steering policy [8], multi-speed functional units come with a criticality-based steering policy [13], and grid-architecture come with a static hyperblock scheduler [11].

Common to these control policies is the goal of attempting to eliminate the performance impact of non-uniformity. The common underlying goal motivates this paper to treat the *non-uniform control* as a problem in its own right. Specifically, we ask: "Should control policies be guided by the same inputs?" and if so, "What might those unifying inputs be?" The answer to these questions may facilitate effective design of future control policies, especially in aggressively non-uniform designs where multiple control policies must coexist.

A common practice is to guide the policies with (ad hoc) design-specific inputs (such as register-dependence information that guides cluster steering). A natural and more general input is the *criticality* of a dynamic instruction. Motivated by the observation that the performance penalty is eliminated if low-quality resources never appear on an execution's critical path, one may use a criticality predictor [4, 13, 16] in an attempt to steer critical instructions to high-quality resources.

Unfortunately, criticality has several limitations. First, criticality does not tell us how many cycles a non-critical instruction can be delayed without impact. Second, criticality partitions instructions into only two classes (critical and non-critical), making it less suitable for multi-way control policies, which are needed when resources are available at more than two quality levels. Third, the relative sizes of the two classes may not match the balance desired by the control policy (typically, 95% of instructions are non-critical, which makes it difficult to obtain, for instance, a 1:1 ratio).

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

To address these deficiencies, we advocate guiding control policies with *slack*. Slack is a concept taken from network analysis [1] and recently applied to microarchitecture [3, 6, 12]. *Intuitively, the slack of a dynamic instruction i is the number of cycles i can be delayed with no effect on the execution.* Slack is inherently more powerful than criticality: since it reveals the "degree of criticality" of an instruction, it enables splitting instructions into more than two classes and tuning their sizes to match the needs of the various non-uniform resources.

Despite its simple definition, slack is a complex phenomenon. Even simply *exploring its potential* warrants closer examination of the definition. In particular, does "no effect on the execution" mean no effect on any dynamic instruction or no effect on the last dynamic instruction? Furthermore, how does one compute slack without having to delay the instruction and observing whether execution time increased? The task of *exploiting slack in practice* poses further challenges. In particular, how does one build a dynamic slack predictor and how should slack be used to guide a policy? This paper makes the following contributions in understanding and exploiting slack:

Modeling and characterizing microarchitectural slack. Section 2 makes three contributions. First, it defines slack formally, using a dependence-graph model that captures data dependences as well as microarchitectural resource constraints [4]. Second, it distinguishes three slack variants—*local*, *global*, and *apportioned*—that are appropriate to different control situations. Finally, a limit study reveals the existence of considerable slack. For example, 75% of dynamic instructions can be delayed by five or more cycles with no impact on program execution time. This result provides encouragement that future control policies may be able to use slack to hide the non-uniformities of emerging constraint-aware designs.

Slack prediction. To apply slack in practice, Section 3 contributes two algorithms for dynamic slack prediction. The first algorithm predicts *explicit* slack, i.e., the actual value of an instruction's slack. The second algorithm, for which we evaluate a hardware design, predicts *implicit* slack, i.e., whether an instruction can tolerate the delay of a particular slow (non-uniform) resource. The predictor effectively matches the slack available in the *micro-execution* with the non-uniformity in the *machine design*, with the goal of hiding non-uniform delays. The predictor is relatively easy to implement, since it consists of only a simple state machine and the token-passing analyzer of Fields, et al. [4].

Application of slack in non-uniform control. In Section 4, we provide an example use of slack as a control mechanism. We show that slack can successfully guide a steering-and-scheduling policy on a non-uniform machine in which some pipelines (including the instruction window, register file, and functional units) run at half the frequency. Specifically, our slack-based policy improves performance on such a machine by up to 20% (10% on average) over the best existing policies, coming within 3% of a higher-power machine with all fast pipelines.

2 Characterizing Microarchitectural Slack

This section presents a study of microarchitectural slack. We explain its nature, measure its amount, and also discuss its implications on what non-uniform microarchitectures it encourages us to build.

The *slack* of an instruction i is the number of cycles i can be delayed without increasing the overall execution time. (Note that in this section, whenever we refer to an *instruction*, we mean a *dynamic instruction*.) Before conducting our experiments, we must carefully refine this seemingly simple definition and develop algorithms for its efficient computation. In the following subsections, we address four main issues:

Modeling microarchitectural slack (Section 2.1): In a complex processor, the impact of delaying an instruction i depends not only on program dependences but also on the resource constraints of the machine. What are the important machine resources to consider when computing slack, and how do we account for them?

Apportioning slack (Section 2.2): When an instruction has slack, that slack can be exploited either by the instruction itself, or by its dependent instructions. In general, the slack can be apportioned among multiple instructions that will be delayed simultaneously. What is a good way to report the amount and "apportioning flexibility" of slack?

Methodology (Section 2.3): To compute an optimal apportioning of slack across multiple instructions, it is necessary to examine large segments of the execution. How do we compute the apportioned slack efficiently?

Analysis (Section 2.4): The amount of slack affects microarchitectural decisions. What are the implications of our empirical observations on non-uniform control policies, and on the non-uniform machines that make sense to build in the future?

2.1 Modeling Slack

In order to experimentally determine *microarchitecturally accurate* slack, we must understand what impact delaying an instruction has on the complex mechanisms of an out-of-order processor—where resource constraints, as opposed to data dependences, sometimes dictate the amount of slack an instruction has. For example, if no instructions data dependent on i are fetched, we may be able to delay execution of i until just before it must be committed to avoid stalling the reorder buffer. We use the term *microexecution* to include all aspects of a given program execution on a given microarchitecture. Understanding microexecutions is important both for measuring slack via offline analysis (which we explore in this section) as well as predicting slack in hardware (discussed in Section 3).

A natural way to account for all microarchitectural effects on slack is to do so indirectly (but accurately), by employing a *delay-and-observe* approach: to deter-

IEEE COMPUTER SOCIETY

mine the slack of an instruction, delay its execution by n cycles and observe if the *overall* execution time is increased. If it is not, the instruction has at least n cycles of slack. There are two serious complications with the practicality of this approach. First, to determine the precise value of slack, one needs to iterate over various values of n for a particular dynamic instruction, potentially restarting the simulation. Second, short of executing the whole program, it is not clear how to determine whether a given value of delay, n, actually slowed down the execution. These two problems make the delay-and-observe approach challenging for computing the slack of *every* dynamic instruction in the program.

Srinivasan, et al. [15] made the delay-and-observe approach feasible by sacrificing some accuracy. To avoid restarting the whole simulation, they equipped their simulator with a capability to roll back to the delayed instruction (which was always a load). To avoid rolling back from afar, they estimated early, using a set of heuristics, whether the delay actually slowed down the entire execution. The heuristics, such as whether the issue rate drops below a threshold, resulted in a measurement error of about 8%. While this methodology provided powerful (and reasonably efficient) analysis of load instructions, it may be difficult to extend its delay-and-observe approach to determining the slack when multiple non-load instructions are delayed (as opposed to a single load).

To avoid the problems with the delay-and-observe approach, our study uses an off-line method based on constructing a dependence-graph model of the execution. The graph is built by the simulator during the execution, with each edge corresponding to a dependence and annotated with the dependence's observed latency. After the execution, the slack is computed by determining how much latencies can be extended without growing the critical path of the graph (see Section 2.3).

Clearly, the graph composed of only *data* dependences will not provide much microarchitectural accuracy. The problem with a data dependence graph (DDG) is that it omits microarchitectural resource constraints, which can severely skew our slack measurements. Consider the extreme case of an instruction that writes a memory location that no other instruction in the program reads. According to the DDG, this instruction may have millions of cycles of slack. Its microarchitecturally accurate slack, however, is much shorter, as delaying the instruction by millions of cycles would stall the reorder buffer and cause a degradation in performance.

Casmira and Grunwald [3] avoid this problem by computing a "scheduling slack," which is the slack observed on a DDG constructed from instructions present in the instruction window each cycle. While this restriction adds a degree of resource sensitivity, it is still conservative in its estimation of slack. For instance, if there is only one instruction in the window, it will be determined to have a slack of zero (as it will be on the critical path), whereas this instruction may in fact have a great deal of slack (if none of its data-dependent instructions are going to be fetched for many cycles).

Fundamentally, a dependence graph is microarchitecturally accurate only when it models all dependences that govern the corresponding processor (or, equivalently, its simulator model). To obtain a microarchitecturally *sensitive* graph model, we use the model of Fields, et al. [4]. The model, summarized in Table 1 and Figure 1, accounts for in-order fetch (with edges from D_i to D_{i+1}), in-order commit (with edges from C_i to C_{i+1}), and out-of-order execution (by constraining pairs of E-nodes only with data-dependence edges). It also models fetch stalls due to the reorder buffer (with edges between C-nodes and D-nodes) and branch mispredictions (with edges between E-nodes and D-nodes, e.g., $E_7 \rightarrow D_8$). Finally, the graph also models functional-unit contention, by adding observed contention cycles into the execution latency (which is placed on *EE* and *EC* edges).

Once we have built the graph, we can identify the amount of slack an instruction has by determining how far it is from the critical path. In Figure 1, all instructions not on the critical path (marked in bold) have some amount of slack.

While this model omits some dependences, (for example, between loads that share a cache line), our validation described in Section 2.4 found that our slack calculation error is only about 1%.

2.2 Apportioning Slack

While the dependence-graph model solves the problem of how to accurately *identify* microarchitectural slack, it leaves open the question of how to *report* the slack available in the graph. The problem is that we want to distribute available slack among potentially many instructions (to be delayed simultaneously) but that distribution will vary depending on the non-uniformity to be hidden.

For example, to quantify the amount of slack available to a *set* of instructions that are to be delayed simultaneously, we define a notion of *apportioned* slack. Before we define apportioned slack, however, we will define *local* slack and *global* slack, which characterize the slack available to an *individual* instruction.

Local slack of a dynamic instruction i is the maximum number of cycles the execution of i can be delayed *without delaying **any** subsequent instructions*. From our measurements, approximately 20% of instructions have local slack greater than five cycles. Local slack is conservative because it prevents delaying any instruction 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 instruction i is the maximum number of cycles the execution of i can be delayed *without delaying **the last** instruction in the program*. From our measurements, approximately 40% of instructions have global slack greater than 50 cycles. In other words, there is a particular set of 40% of all instructions of which *one* instruction can be picked and delayed by 50 cycles without increasing execution time. Global

IEEE COMPUTER SOCIETY

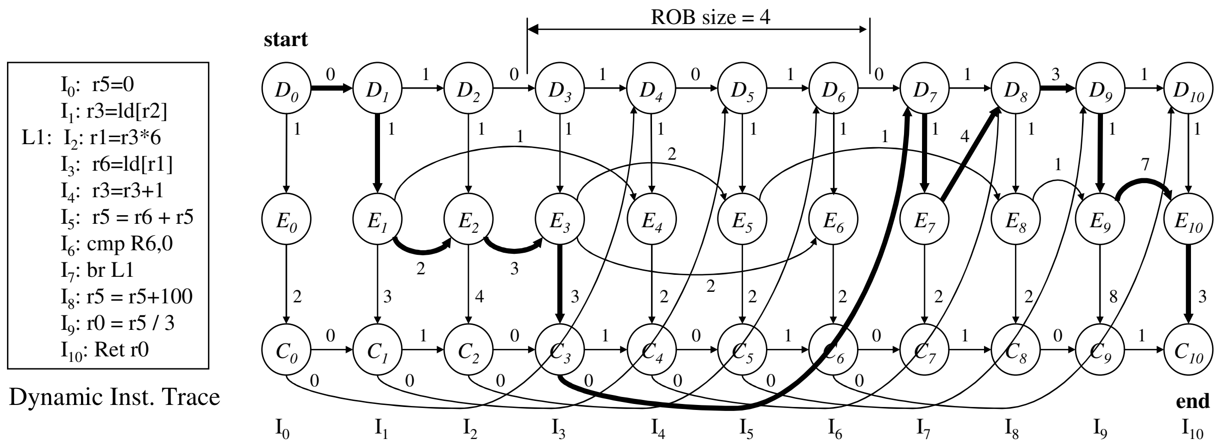


Figure 1: **An instance of the critical-path model from Table 1.** The dependence graph represents a sequence of dynamic instructions. Nodes are events in the lifetime of an instruction (the instruction being dispatched, executed, or committed); the edges are dependences between the occurrences of two events. A weight on an edge is the latency to resolve the corresponding dependence. The critical path is in bold.

name	constraint modeled	edge
DD	In-order dispatch	$D_{i-1} \rightarrow D_i$
CD	Finite re-order buffer	$C_{i-w} \rightarrow D_i$ [a]
ED	Control dependence	$E_{i-1} \rightarrow D_i$ [b]
DE	Execution follows dispatch	$D_i \rightarrow E_i$
EE	Data dependences	$E_j \rightarrow E_i$ [c]
EC	Commit follows execution	$E_i \rightarrow C_i$
CC	In-order commit	$C_{i-1} \rightarrow C_i$

[a] $w =$ size of the re-order buffer
[b] inserted if $i - 1$ is a mispredicted branch
[c] inserted if instruction j produces an operand of i

Table 1: **Dependences captured by the critical-path model**, grouped by the target of the dependence.

slack thus reflects a policy that would seek to delay one instruction by many cycles. Global slack also 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.

Apportioned slack captures slack available when we desire to delay *multiple instructions simultaneously*. Namely, we want to determine how many instructions can be delayed together by a certain amount of slack without impacting the execution. The desired amount of delay for each instruction depends on the apportioning strategy, which in turn depends on the particular non-uniformity whose latency we seek to hide. Thus, while global slack indicates how much *one instruction* can be delayed, apportioned slack indicates how much *a set of instructions* can be delayed simultaneously.

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 strategy. Next, we define two such strategies we use later in our experiments.

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 (described in Section 2.4), 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 suggest tremendous optimization opportunities.

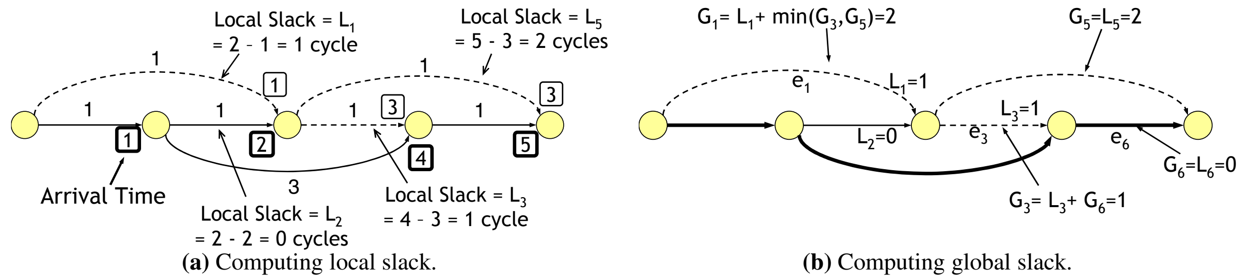
Latency-plus-one-cycle apportioning. Another apportioning strategy that we consider reflects a control policy for a constraint-aware processor that has a (power-efficient) ALU that runs at half the frequency of the other ALU. The goal of the control policy would be to maximize the number of instructions 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).

2.3 Algorithms for Calculating Slack

Next, we outline the algorithms for efficiently computing the three variants of slack on the dependence graph constructed during the simulation. For simplicity, we illustrate our algorithms using simple dependence graphs where *each node is a dynamic instruction*, but our experimental results use the graph of Section 2.1.

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

1063-6897/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY



(a) Computing local slack.

(b) Computing global slack.

Figure 2: **Computing local and global slack.** Local slack is computed as the difference in arrival times of incoming edges. Global slack is computed via a reverse topological sort. Edges with nonzero local slack are dashed. In **(b)**, the critical path is in bold.

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 2(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.

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 2(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. This overall computation is a simple, linear time, reverse topological sort.

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. Due to space constraints, we only sketch the algorithm here, using the five-cycle apportioning strategy for illustration. Basically, the backward global-slack pass accumulates local slacks and deposits them on the earliest possible nodes, from where it is picked up by the forward apportioning pass. As the forward pass encounters each node v, it is decided whether enough global slack exists to apportion v five cycles of slack. If enough exists, v is apportioned five cycles, and it is ensured that no other nodes further downstream are apportioned those five cycles. This process continues until the forward pass reaches the end of the program.

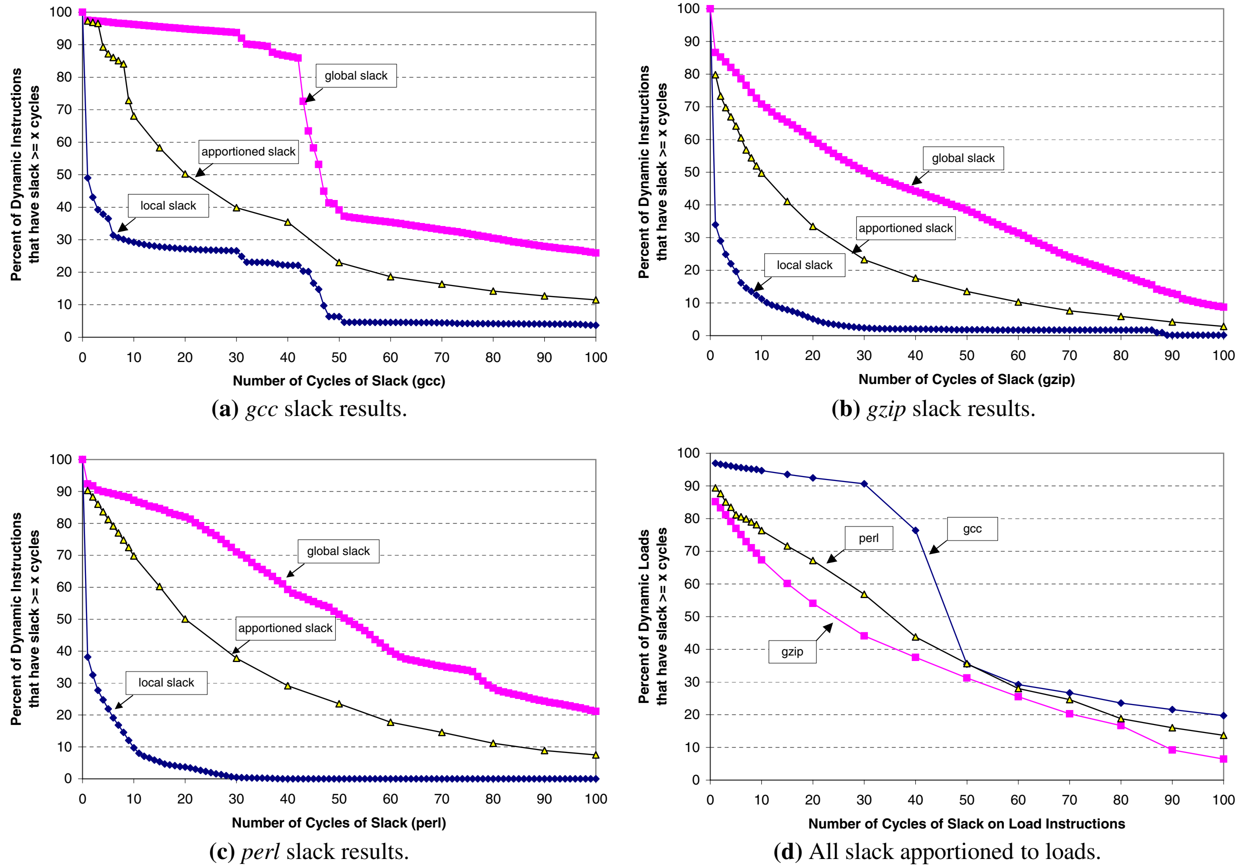
2.4 Experimental Characterization of Slack

This section presents experimental characterization of local, global and apportioned slack. Our results show that slack has a tremendous potential for hiding non-uniform latencies, in particular when large local slacks are apportioned to multiple instructions across dependence chains. This section also addresses the implications of slack: we discuss what types of design non-uniformities can be tolerated with slack and what cannot. Finally, we validate our methodology, demonstrating that our findings are very accurate.

First, we explore the *amount* of available slack, focusing on microexecutions of typical SPEC2000 programs on an unclustered version of the 6-wide processor described in Section 4.2. We compute slack using the graph of Section 2.1, and when we refer to the slack of an instruction, we mean the slack of that instruction's E node. Figures 3(a)-3(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 cycles. 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.

1063-6897/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY



(a) *gcc* slack results. **(b)** *gzip* slack results. **(c)** *perl* slack results. **(d)** All slack apportioned to loads.

Figure 3: **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.

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 3(a)-3(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).

1063-6897/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

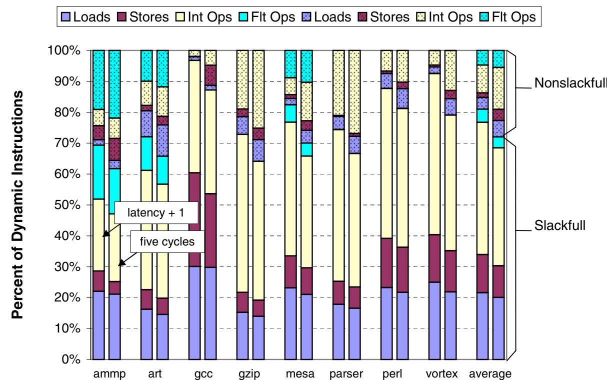


Figure 4: **Limit studies.** Measurements for two apportioning strategies are shown: *latency-plus-one-cycle* and *five-cycle* apportioning. These measurements provide an indication as to what types of non-uniform machine designs can be tolerated by a slack-based policy. For instance, latency-plus-one-cycle apportioning is relevant for the fast/slow pipeline microarchitecture studied in Section 4.

Of course, the above apportioning strategy does not reflect all 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 [11]. To maximize slack on loads, we modify the above apportioning strategy such that no slack is apportioned to non-load instructions. Figure 3(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 4, 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 2.2: 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 4 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 Section 4.

- 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.

Validation. We need to validate our experiments since (as previously mentioned) the dependence-graph model we use to compute slack only includes the most significant microarchitectural dependences. Thus, the slack measurements have some error.

We confirm correctness of the slack measurements by the following two-step process: (1) we identify apportioned slack on the graph, as usual; and, then, (2) we rerun the simulation on which the graph was constructed, but in this new run, each dynamic instruction is delayed by its apportioned slack. Since we are delaying the instruction in the actual (simulated) execution, errors in the graph-computed slack will be manifested as increases in the execution time of the simulation.

We performed this validation with several different apportioning strategies: *latency-plus-one-cycle* and *five-cycle* apportioning from Figure 4 and *12-cycles to loads* from Figure 3(d). Space limitations prohibit detailed presentation of results, but the maximum error observed across all benchmarks and apportioning strategies was less than 3%, with an average error of about 1%, which is less than previous related efforts [15].

3 Predicting Slack

Our slack predictors follow the history-based approach used in most hardware predictors: the slack of a dynamic instruction, known after the instruction commits, trains a PC-indexed predictor, which is then used to predict the slack of future instances of the same static instruction. An alternative would be a context-based approach that would predict slack based on the current state of, say, the scheduling window [5]. The advantage of the history-based approach is that it allows predicting slack early in the pipeline.

Two conditions must be met to enable history-based slack prediction. First, there must be a locality of slack, in that dynamic instances of a given static instruction exhibit roughly the same slack. Second, we must design a hardware mechanism for measuring slack of a dynamic instruction. We meet the two conditions in the following two subsections. Then, equipped with a hardware slack *detector*, we develop two slack *predictors*. The predictors differ in what is stored in the predictor table: the *explicit-slack* predictor learns the actual value of slack of the static instruction; the *implicit-slack* predictor learns whether the static instruction can tolerate the delay of a particular non-uniform resource.

3.1 Locality of Slack

Since slack arises partly due to microarchitectural events, like reorder-buffer stalls caused by cache misses, one might expect that dynamic slack is distributed across the instruction stream more or less randomly, complicating

IEEE COMPUTER SOCIETY

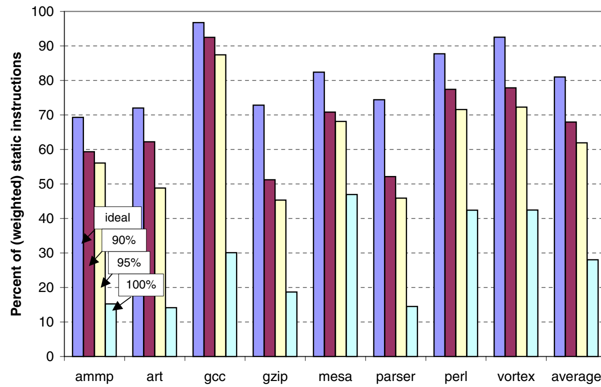


Figure 5: **Mapping dynamic slack behavior to static instructions.** Uses latency-plus-one-cycle apportioning. On the y-axis, the number of slackful static instructions is weighted by the number of each static instruction's dynamic instances.

predictability. 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 5). 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).

In more detail, our experiments used the following methodology. First, we computed the apportioned slack using the latency-plus-one strategy introduced in Section 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 5 plots the amount of slackful static instructions. 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.

3.2 Measuring Slack in Hardware

In Section 2.1, we described the delay-and-observe approach, as a natural—but expensive approach—for accurately measuring slack in a processor *simulator*. In that approach, a dynamic instruction is delayed by n cycles, after which the execution is observed. If the overall execution is not slowed down, the instruction has at least n cycles of (global) slack.

In this section, we use the delay-and-observe paradigm to design *hardware* for measuring slack. We accomplish this goal by elegantly solving the two implementation challenges of the delay-and-observe approach. Specifically, the challenges are: **(1)** measuring slack of a dynamic instruction requires repeatedly delaying the instruction for various values of delay, which involves rolling back the execution; and **(2)** determining (naively) whether the overall execution was affected by the delay requires comparing the original and the perturbed execution. To solve the first challenge, we sample each *dynamic* instruction at most once. Such sampling avoids the need for rollback yet is sufficient to determine the slack of a *static* instruction, since we exploit the locality of slack presented in Section 3.1.

To solve the second problem (determining whether the execution was affected by the delay), we exploit the following observation: *the overall execution is slowed down by the delay if and only if the delayed instruction appears on the critical path of the micro-execution.*[1] With this observation, we can reduce the problem of detecting slack to that of determining criticality, which can be easily performed using our token-passing criticality detector [4].

For the sake of completeness, we sketch here the algorithm behind the token-passing detector of criticality [4]. For simplicity, we will explain its operation on data dependences, but the detector actually operates on the graph model illustrated in Figure 1. The detector is based on the observation that a dynamic instruction i is not critical if either of two conditions hold: **(1)** the value v computed by i arrives at each consumer j before one of j's remaining operands arrives (*i.e.*, if v is not *last-arriving* and, hence, has non-zero local slack at j); or **(2)** if the consuming instruction j is not critical. Thus, we need to determine if the value v computed by i traverses a long chain of data dependences where consumers are always waiting for it (*i.e.*, where it was always last arriving). If this situation occurs, i is predicted critical—otherwise it is known to be non-critical.

This observation lends itself to an efficient hardware implementation: to determine if dynamic instruction i is critical, plant a token into i (the token can be thought of as an extra bit appended to the data computed by the instruction). The token is then propagated together with the data to all dependent instructions, except that it is killed whenever the data is not last-arriving at a consumer instruction. After a few hundred instructions, the detector examines the machine: if at least one copy of the token is alive, the dynamic instruction was critical (because there must have been at least one chain of data dependences on which the data was always last arriving.) With high probability, i is part of the critical path, since delaying i would delay all instructions on this long chain.

To put the pieces together, our slack detector works as follows. Given a dynamic instruction i and a value n, the slack detector answers the question "does i have at least n cycles of slack?" The slack detector is a simple delay-and-observe extension of the criticality detector [4]: it first delays the instruction by $n-1$ cycles (see footnote 1) and then observes, by planting a token into i, whether the delayed instruction i is critical. If i is critical, then it does not have n cycles of slack; otherwise it does.

[1] Strictly speaking, this observation makes a one-cycle mistake, because a delay may make the dynamic instruction critical without making the critical path any longer.

1063-6897/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

3.3 Explicit-Slack Predictor

The explicit slack predictor learns and predicts the *actual* number of cycles of slack available in each static instruction. The predictor is trained by sampling dynamic instructions—using the slack detector described above—under various values of slack n. The goal of sampling is to converge to the average value of slack for each static instruction, which can be achieved with a binary search (assuming the instruction has good slack locality).

It should be noted that training the predictor by delaying a dynamic instruction is not likely to noticeably slow down the program because **(i)** sampling is sparse (in our designs, the sampling rate is roughly 1 instruction per 100 instructions, and can be even sparser), and **(ii)** the inserted delay is typically just large enough to make the instruction critical, which means that the delay may extend the critical path by at most a few cycles.

Predicting explicit slack, however, produces some challenges. Most importantly, it is not clear how to avoid measurement perturbation due to non-uniform 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.

3.4 Implicit-Slack Predictor

We address the measurement perturbation problems with an *implicit-slack* predictor, which, instead of predicting *exactly* how much slack an instruction has, predicts whether it has *enough* slack to tolerate a particular non-uniform resource—for instance, whether its execution latency can be doubled without impact on performance.

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 the non-uniform machine used in Section 4 for our experiments. 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* non-uniform resources, where each dynamic instruction is assigned to one resource. In general, if a control policy must make k decisions for each instruction, we have 2^k virtual resources, each corresponding to a slack bin.

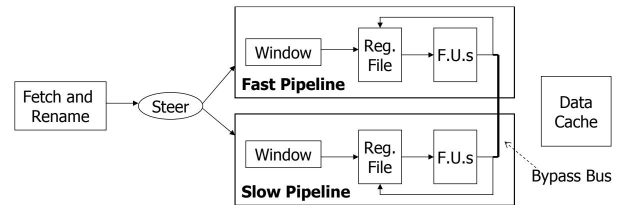


Figure 6: **The non-uniform microarchitecture used in our experiments.** The processor consists of one fast and one (or two) slow pipelines.

Measuring implicit slack has four important advantages. First, when sampling the slack, we don't need dedicated logic to artificially delay the instruction. Instead, the predictor can delay the instruction *naturally*, by steering it to the sampled non-uniform resource. Second, as desired, by measuring tolerance to non-uniform delays, we effectively remove the impact of perturbation on the measurement. Third, bin membership can be trained faster than the actual slack. Dealing with bins, rather than with the actual slack, can be much easier for the control policy.

Finally, it should be noted that while the four slack bins above are ordered in seemingly decreasing priority, it does not mean that the slack of instructions in bin 3 is greater than those in bin 2. In general, into which bin an instruction falls depends purely on which resource it can tolerate, which is the fourth advantage of the the implicit-slack approach.

4 Example Use of Slack in Non-Uniform Control

In this section, we evaluate the success of slack in guiding a non-uniform control policy. We define an aggressively non-uniform (power-aware) microarchitecture and design a slack-based control policy for hiding its non-uniformities. We compare the 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.

Specifically, we evaluate a slack-based control policy on the machine pictured in Figure 6. 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 functional-unit speed.)

1063-6897/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

4.1 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 physical distance and 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 3.4. 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 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 3.4) 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 level.

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 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. [16], as a replacement for the token-passing criticality analyzer [4] in the slack detector (see Section 2.4). (We also experimented with the QOLD criticality predictor from the same work [13, 16], but the ALOLD predictor performed considerably better in our context.)

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

4.2 Methodology

Our evaluation uses a typical dynamically-scheduled superscalar processor as a baseline whose configuration is detailed in Table 4. The simulator is built upon the SimpleScalar tool set [2]. Our benchmarks consist of a subset of the SPEC2000 benchmark suite; all are optimized Alpha binaries using reference inputs. Initialization phases were skipped and detailed simulation ran until 100 million instructions were committed.

4.3 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, 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 ad-

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 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.

Dynamically Scheduled Core	128-entry instruction window (64 entries in each of 2 pipelines) with critical-first scheduling, 256-entry re-order buffer, 6-way issue, 12-cycle pipeline, perfect memory disambiguation, fetch stops at second taken branch in a cycle, 1 cycle normal bypass latency plus one cycle extra delay if sending data from one clock domain to another.
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	64KB 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)	In each of 2 pipelines: 3 Integer ALUs (1), 1 Integer MULT (3), 2 Floating ALU (2), 1 Floating MULT/DIV (4/12), 1 LD/ST ports (2).
Token-passing Slack Predictor	4K-entry array for storing predictions (2 bit bin, 6 bit hysteresis per entry), 768-byte training array—(8 tokens x 3 nodes x 256-entry ROB) bits

Table 4: **Configuration of simulated processor.**

dition 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 two additional configurations were *3f+3s+3s* and *Half_3f+3s+3s*, each with one 3-wide fast pipeline and two 3-wide slow pipelines; but, in *Half_3f+3s+3s*, the window size of each slow pipeline is halved (so that the effective window size is equal to that of *3f+3f*). The decrease in window size of *Half_3f+3s+3s* resulted in only a modest performance loss of 1–2% compared to *3f+3s+3s*,

To estimate the power savings obtained from the configurations, we can directly apply the relationship $P \propto F^2$. For the *3f+3s* configuration, we save 37.5% of the power of the *core* (including the instruction window, issue logic, register file, and functional units), and for the *3f+3s+3s* configuration, we save 25%. The latter result is interesting, since it suggests we can obtain significant power savings with some cost in area, but no loss in performance, by exploiting a control policy based on slack.

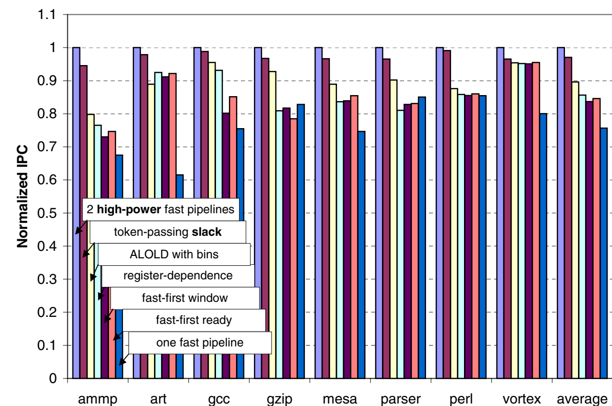


Figure 7: **Comparing control policies on fast/slow pipeline microarchitecture.** All measurements are normalized to the baseline of two fast 3-wide pipelines (*3f+3f*). Also, results are shown for a single fast 3-wide pipeline (*3f*) for reference. The rest of the measurements are different control policies for a *3f+3s* machine.

5 Related Work

As most related work has already been discussed in relevant sections, we will only summarize here. Srinivasan and Lebeck [15] and Rakvic, et al. [10] perform measurements of load latency tolerance in out-of-order processors. The concept of using dynamic scheduling slack for controlling microarchitectures through clustered voltage scaling was proposed by Casmira and Grunwald [3]. In these works, no slack predictor design was studied.

Much research has explored using critical-path predictions in control policies for various optimizations, including power optimizations [9, 13], cluster steering [4, 16], dynamic instruction scheduling [4], value prediction [4, 16], and cache optimizations [5, 14]. In our work,

IEEE COMPUTER SOCIETY

we characterize and exploit the more powerful property of *slack* and show how to predict it. We show how to exploit prior research into criticality predictors when designing an efficient slack predictor, principally by “binning” instructions based on the latency they can tolerate.

Semeraro, et al. [12] use 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. Our work characterizes slack more fully and provides an online predictor. Grunwald [6] describes a hardware predictor based on measuring how much an instruction’s execution can be delayed without delaying subsequent instructions. This hardware appears to predict what we call local slack in our work. As shown in Section 2, local slack is only a small part of all the slack that is available.

The model and token-passing criticality detector we used came from our previous work on critical-path prediction [4]. We have extended this research to measure and predict slack, as opposed to simple criticality, and showed how slack can be exploited to hide the latencies of nonuniform machine designs.

6 Conclusion

We have developed slack as a useful input for guiding control policies in modern processors. We defined three variants of slack (having in mind various applications of slack) and presented a novel methodology for accurately measuring the amount of slack available in programs. We have shown there is a surprisingly large amount of exploitable slack and that most of it can be predicted easily with a token-passing criticality analyzer.

Finally, we showed how to design a slack-based control policy for a power-efficient microarchitecture with fast and slow pipelines. Our experiments showed that the slack-based policy eliminates most of the penalty due to the non-uniformities, such that the fast/slow pipeline microarchitecture performs nearly as well as a uniform machine with only fast pipelines. This experiment has significant implications for future machine designs: we may be able to mitigate technological constraints (e.g., wire delay, power, and circuit complexity) by building non-uniform machines and then controlling them with a slack-based policy.

Acknowledgements. We thank Alvy Lebeck, Sanjay Patel, Nilofer Motiwala, Paramjit Oberoi, Shai Rubin, Craig Zilles, and the anonymous reviewers for comments on drafts of the paper. We also thank Konrad Lai, C.J. Newburn, Jared Stark, Srikanth Srinivasan, Chris Wilkerson, and the Wisconsin Architecture Industrial affiliates for discussions on slack and its exploitation. This work was supported in part by an IBM Faculty Partnership Award, National Science Foundation grants (EIA-0103670, EIA-9971256, CDA-9623632, and CCR-0105721), an NSF CAREER award (CCR-0093275), a Wisconsin Romnes Fellowship, and donations from Intel, Microsoft, and Sun Microsystems. Brian Fields was partially supported by an NSF Graduate Research Fellowship.

References

[1] A. Battersby. *Network Analysis for Planning and Scheduling, 3rd. ed.* John Wiley and Sons, Inc., 1970.

[2] 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.

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

[4] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Jun–Jul 2001.

[5] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *IEEE International Conference on Computer Design*, Oct 1999.

[6] D. Grunwald. Chapter 4: Micro-architecture design and control speculation for energy reduction. *To Appear in Power-Aware Computing*, edited by R. Graybill and R. Melhelm, Kluwer Academic Publishers.

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

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th Annual International Symposium on Computer Architecture*, Jun 1997.

[9] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *Workshop on Complexity-Effective Design in conjunction with ISCA 2001*, Jun 2001.

[10] R. Rakvic, D. Limaye, and J. P. Shen. Non-vital loads. Technical Report CMuART-2000-02, Carnegie Mellon University, 2000.

[11] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec 2001.

[12] 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 *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, Feb 2002.

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

[14] S. T. Srinivasan, R. Dz ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Jun 2001.

[15] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Nov–Dec 1998.

[16] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001.

1063-6897/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY