A Mechanistic Performance Model for Superscalar Out-of-Order Processors

STIJN EYERMAN and LIEVEN EECKHOUT
Ghent University
TEJAS KARKHANIS
Advanced Micro Devices
and
JAMES E. SMITH
University of Wisconsin – Madison

A mechanistic model for out-of-order superscalar processors is developed and then applied to the study of microarchitecture resource scaling. The model divides execution time into intervals separated by disruptive miss events such as branch mispredictions and cache misses. Each type of miss event results in characterizable performance behavior for the execution time interval. By considering an interval's type and length (measured in instructions), execution time can be predicted for the interval. Overall execution time is then determined by aggregating the execution time over all intervals. The mechanistic model provides several advantages over prior modeling approaches, and, when estimating performance, it differs from detailed simulation of a 4-wide out-of-order processor by an average of 7%.

The mechanistic model is applied to the general problem of resource scaling in out-of-order superscalar processors. First, we use the model to determine size relationships among microarchitecture structures in a balanced processor design. Second, we use the mechanistic model to study scaling of both pipeline depth and width in balanced processor designs. We corroborate previous results in this area and provide new results. For example, we show that at optimal design points, the pipeline depth times the square root of the processor width is nearly constant. Finally, we consider the behavior of unbalanced, overprovisioned processor designs based on insight gained from the mechanistic model. We show that in certain situations an overprovisioned processor may lead to improved overall performance. Designs where a processor's dispatch width is wider than its issue width are of particular interest.

T. Karkhanis is now with IBM TJ Watson. J. E. Smith is now with Intel Corporation.
S. Eyerman and L. Eeckhout are Postdoctoral Fellows with the Fund for Scientific Research-Flanders (Belgium) (FWO-Vlaanderen). Additional support was provided in part by the FWO projects G.0232.06 and G.0255.08, and the UGent-BOF project 01J14407.
Authors' addresses: S. Eyerman, L. Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium; T. Karkhanis, Advanced Micro Devices, One AMD Place, P.O. Box 3453, Sunnyvale, CA 94088-3453; J. E. Smith, ECE Department, University of Wisconsin – Madison, 2359 Engineering Hall, 1415 Engineering Drive, Madison, WI 53706-1691.

DOI 10.1145/1534909.1534910 http://doi.acm.org/10.1145/1534909.1534910

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; C.1.3 [**Computer Systems Organization**]: Other Architecture Styles—*Pipeline processors*

General Terms: Performance

Additional Key Words and Phrases: Superscalar out-of-order processor, analytical modeling, performance modeling, mechanistic modeling, resource scaling, pipeline depth, pipeline width, wide front-end dispatch processors, balanced processor design, overprovisioned processor design

ACM Reference Format:

Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. 2009. A mechanistic performance model for superscalar out-of-order processors. ACM Trans. Comput. Syst. 27, 2, Article 3 (May 2009), 37 pages. DOI = 10.1145/1534909.1534910 http://doi.acm.org/10.1145/1534909.1534910

1. INTRODUCTION

For studying superscalar out-of-order processors, both researchers and designers rely heavily on detailed simulation. Although detailed simulation provides accurate feedback regarding specific design configurations, individual simulations give relatively little insight into the fundamental interactions that take place within the processor. It requires a very large number of detailed simulations to capture trends and relationships involving microarchitecture parameters and applications. In this article we propose a mechanistic model, *interval analysis*, as a better method for gaining insight into superscalar out-of-order processors and for guiding higher level design decisions.

Figure 1 illustrates a typical superscalar out-of-order processor with some of the key microarchitecture resources identified. Most readers will readily recognize the primary resources, and Section 2 will discuss them in a more detail. In its current formulation, interval analysis focuses on the flow of instructions through the dispatch stage of a superscalar out-of-order processor.[1] In this article, *dispatch* refers to the movement of instructions from the front-end pipeline into the reorder and issue buffers. The basis for interval analysis is the observation that, in the absence of miss events such as branch mispredictions and cache misses, a well-balanced superscalar out-of-order processor should smoothly stream instructions through its pipelines, buffers, and functional units. Under ideal conditions the processor sustains a level of performance (instructions per cycle) roughly equal to the superscalar dispatch bandwidth. However, the smooth dispatch of instructions is intermittently disrupted by miss events. The effects of these miss events at the dispatch stage divide execution time into intervals, and these intervals serve as the fundamental entity for analysis and modeling; see Figure 2. Interval analysis thereby provides a way of visualizing the significant performance events that take place in an out-of-order processor without requiring detailed tracking of extremely large numbers of individual instructions. Both pipelined instruction processing and the interactions among instructions (e.g., due to data dependences) are modeled at a statistical level

[1]In previous versions [Karkhanis and Smith 2007, 2004; Eyerman et al. 2006a, 2006b], we focused on the issue stage, but have since found that focusing on the dispatch stage leads to a simpler formulation.

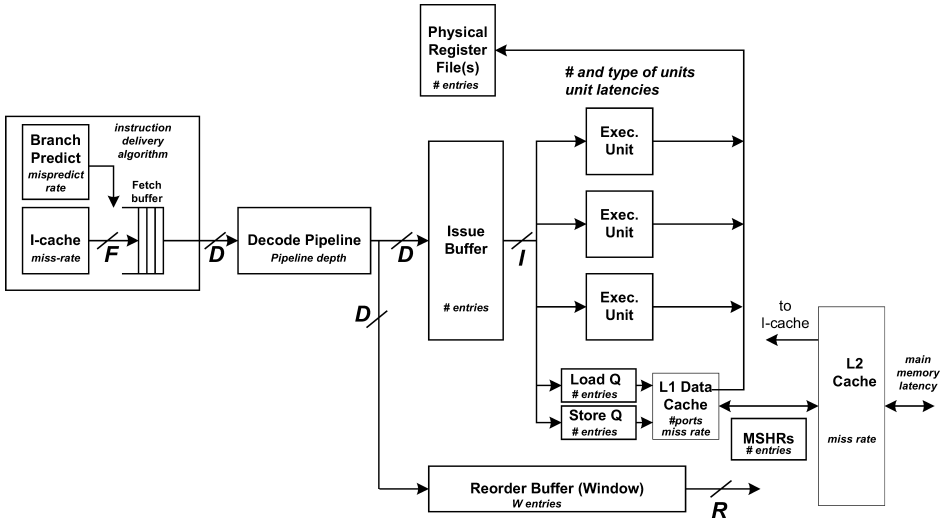

Fig. 1. A parameterized superscalar out-of-order processor.

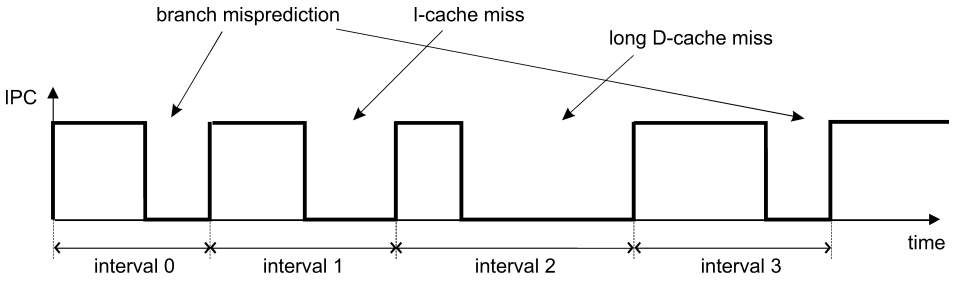

Fig. 2. Performance can be analyzed by dividing time into intervals between miss events.

where the average behavior of key performance-related phenomena provides the needed insight.

1.1 Mechanistic Processor Modeling

Interval analysis is a *mechanistic model*; that is, it is derived from the actual mechanisms in the processor that lead to performance-related behavior. A mechanistic model has the advantage of directly displaying the performance effects of individual, underlying mechanisms, expressed in terms of program characteristics and machine parameters, such as Instruction-Level Parallelism (ILP), misprediction rates, processor width, and pipeline depth. The mechanistic structure is in contrast to the more common empirical models which use machine learning techniques to empirically infer a performance model [Lee and Brooks 2006; Joseph et al. 2006b; Ipek et al. 2006] or hybrid mechanistic/empirical modeling which begins with high-level parameterized equations that have parameters adjusted to "fit" results of detailed, cycle-accurate microarchitecture simulation [Hartstein and Puzak 2002]. Empirical and hybrid

approaches typically lump together program characteristics and/or microarchitecture effects, often characterizing multiple effects with a single parameter. Our proposed mechanistic model, in contrast, is built up from internal processor structure and does not need detailed processor simulation to fit or infer the model; however, we do use detailed simulation to demonstrate the accuracy of the model after it has been constructed.

Using the interval model for superscalar out-of-order processor performance estimation with the SPEC CPU2000 integer benchmarks as a workload, we find that estimated performance differs from detailed simulation by about 7% on average for a 4-wide out-of-order processor. In addition, we demonstrate that across a wide processor design space the mechanistic model closely tracks performance estimates from detailed simulation.

1.2 Resource Scaling of Superscalar Out-of-Order Processors

An important consideration when designing an out-of-order processor is the way that the various structure sizes and bandwidths scale with respect to one another. To demonstrate the usefulness of the mechanistic model, we apply it to study resource scaling in out-of-order processors. Because the various resources and instructions interact in complex ways, a structural, or mechanistic, model is very useful for understanding the fundamental relationships between the instruction stream and these interacting resources. In particular, we do the following.

—We discuss scaling relationships among front-end and back-end processor resources and pipeline depth. We reenforce the already known fact that in a balanced processor design most of the back-end buffer resources (e.g., issue buffer, ROB) scale quadratically (at least) with the pipeline width. In addition, we provide the new insight that these back-end buffer resources also scale quadratically (at least) with pipeline depth in a balanced processor design.

—We study scaling relationships between pipeline depth and performance. Superscalar pipeline depth scaling has been extensively studied or over two decades [Kunkel and Smith 1986; Emma and Davidson 1987; Dubey and Flynn 1990; Agarwal et al. 2000; Hrishikesh et al. 2002; Sprangle and Carmean 2002; Srinivasan et al. 2002; Hartstein and Puzak 2003, 2002]. However, our model breaks pipeline depth scaling into finer components so that the scaling effects of specific types of miss events are apparent.

—We study scaling relationships between pipeline width and performance, as well as the relationship between pipeline width and depth. These relationships have not been widely studied previously.

—We show that for maintaining optimal designs, simultaneously scaling toward wider and deeper pipelines are in opposition. In fact, we derive the result that for a combination of optimum pipeline depth and width, the depth times the square root of the width is nearly constant. For optimality, wider pipelines imply shallower pipelines.

—We show that some overprovisioned processor designs (designs that are not balanced by our definition) may provide performance improvements due to transient conditions caused by miss events. For example, there is an overall performance advantage in having a front-end dispatch width greater than the issue width, and we study this configuration in some detail.

1.3 Article Outline

The article is organized as follows. We first summarize superscalar out-of-order processor microarchitecture in Section 2. Section 3 then discusses and validates the mechanistic model in detail. Subsequently, in Section 4, we use the model for studying resource scaling of out-of-order processors. Finally, we conclude in Section 5.

2. SUPERSCALAR OUT-OF-ORDER PROCESSORS

2.1 A Parameterized Superscalar Processor

We consider superscalar out-of-order processors as illustrated in Figure 1. This is a generic design, similar in style to many processors in use today. The processor incorporates a number of microarchitecture parameters that, along with program-related characteristics, determine overall performance. Some design elements provide performance proportional to size and capacity (e.g., a cache size or a number of buffer entries), others provide performance as a function of configuration (e.g., cache associativity), and for others performance is a complex combination of the two.

A key set of microarchitecture parameters are bandwidths, or simply *widths*, of certain superscalar pipeline stages. These widths are measured in units of Instructions Per Cycle (IPC). Referring to Figure 1, these are the fetch width F, dispatch width D, issue width I, and retire width R. It is assumed that the front-end decode and rename pipeline stages match the dispatch width D.

At the left side of Figure 1 is the instruction delivery subsystem which combines elements of instruction fetch, instruction buffering, and branch prediction. It is important that the instruction delivery subsystem provides a sustained flow of instructions that matches the capacity of the rest of the processor to issue, execute, and commit instructions. Depending on the required instruction delivery rate, the fetch unit may be designed with a certain level of aggressiveness; for example, it might fetch up to the first branch, to the first not-taken branch, or past multiple branches, both taken and not taken [Michaud et al. 1999]. Because of the variability in software basic block sizes, the peak fetch width F will typically be greater than the front-end pipeline width D with an instruction fetch buffer to moderate the flow between fetch and the pipeline front-end. In this work, we assume that the instruction delivery subsystem is aggressive enough to achieve a sustainable fetch rate F_{eff} that is at least as large as the front-end pipeline width D in the absence of front-end miss events; that is, $F_{eff} \geq D$, which is common in contemporary out-of-order microprocessors.

After instructions are fetched, they are decoded, their registers are renamed, and they are dispatched simultaneously into an issue buffer and a ReOrder Buffer (ROB). We define the number of instructions in the ROB to be W, the window size. Note that in this work it is the ROB that defines the "window," not the issue buffer as is sometimes done. In most modern out-of-order processors, the issue buffer is separate from the ROB. The role of the ROB is to maintain the architected instruction sequence of all active or "in-flight" instructions so that the correct process state can be restored in the event of a trap or a branch misprediction. The issue buffer contains the nonissued instructions: the instructions waiting for their dependences to be cleared so that they can issue.

The processor in Figure 1 has a single, unified issue buffer. However, many out-of-order processors have partitioned issue buffers, with one issue buffer feeding a subset of functional units, that is, a floating point issue buffer, an integer ALU issue buffer, a load/store issue buffer, etc. In these processors, the total issue width I is typically larger than the dispatch width D. The reason for having $I > D$ is to achieve an effective instruction issue width I_{eff} equal to the dispatch width D, accounting for variations in the distribution of instruction types. As such, except when stated otherwise, we assume an effective issue width I_{eff} and retire width R are as large as the dispatch width D, and refer to the "processor width" collectively as D.

The pipeline *depth* is a measure of the number of stages required to perform a given function. In Figure 1 all the functions have a depth, even though only the decode pipeline is labeled as such. In a deeper pipeline, more stages are used, or, conversely, there is less logic per stage. Therefore, in a deeper pipeline the clock frequency is usually higher than in a shallow pipeline.

2.2 Balanced Processor Designs

The notion of a processor design being balanced has intuitive appeal, but an actual definition of a balanced design is seldom, if ever, given. The large number of microarchitecture features makes such a definition difficult, and a number of definitions are possible. In this subsection, we define balanced processors in a way that is both intuitive and useful for our processor modeling work.

We define an out-of-order processor design to be *balanced* if, for a given dispatch width D, the ROB (window size) and other resources such as the issue buffer(s), load/store buffers, rename registers, MSHRs, functional units, write buffers, etc., are of sufficient size to achieve sustained processor performance of D instructions per cycle *in the absence of miss events*. Furthermore, for a given balanced design, reducing the size of any one of the resources will reduce sustained performance below D instructions per cycle.

This definition makes intuitive sense because making any of the individual resources larger, that is, overprovisioning it, will lead to a waste of that resource under ideal (no miss event) conditions. And, conversely, underprovisioning one resource will lead to waste of other resources. An important point is that balance is defined in the absence of miss events, so some aspects of a design such as cache and predictor sizes and configurations do not come into play directly. Miss rates vary dramatically depending on applications, so if miss events

were included in the definition, the notion of balance would become much more application dependent.

Not all balanced designs are claimed to be optimal in the presence of miss events. For example, if the cache and predictor structures are so small that miss events are very frequent and the maximum dispatch rate is never achieved, then even a balanced design wastes resources. However, for the same small cache and predictor there is likely to be a smaller (narrower width) balanced design that is a better design point with similar performance at lower cost. A corollary is that, in the presence of miss events, a design with some overprovisioning may lead to improved overall performance, as will be shown in Section 4.

The relationship between dispatch/issue width and window size has long been known to display approximate quadratic behavior [Riseman and Foster 1972; Wall 1991; Michaud et al. 1999]. We will discuss this relationship in more detail and provide supportive data in Section 3.2, but for now, an important summary observation is that this relationship holds for dispatch and issue widths up to at least eight. Hence, a balanced microarchitecture up to a width of at least eight can be modeled with the techniques we present here, and our results extend at least to processors of that size. Note we are not saying that a processor of width eight is necessarily feasible for a given technology. What we are interested in here is a model that spans a large design space, and we simply argue that, based on instruction dependence relationships, the model will span a design space at least as large as width eight.

3. MECHANISTIC PROCESSOR MODELING

The mechanistic processor model is built on a combination of interval analysis and an ILP model, which we discuss in this section.

3.1 Interval Analysis

As described earlier, and illustrated in Figure 2, the basis for interval analysis is that under optimal conditions, a balanced superscalar out-of-order processor sustains Instructions Per Cycle (IPC) performance roughly equal to its dispatch width D. However, when a miss event occurs, the dispatching of useful instructions eventually stops. There is then a period when no useful instructions are dispatched, lasting until the miss event is resolved, and then instructions once again begin flowing.

In Figure 2, the number of useful instructions dispatched per cycle (IPC) is shown on the vertical axis and time (in clock cycles) is on the horizontal axis. We specifically say "useful" instructions to exclude speculative instructions that are later discarded due to a branch misprediction. As illustrated in Figure 2, the effects of miss events divide execution time into intervals. Intervals begin and end at the points where instructions just begin dispatching following recovery from the preceding miss event. In other words, an interval includes the time period following the particular miss event when no instructions are dispatched.

An interval consists of two parts, as illustrated in Figure 3. The first part performs useful work in terms of dispatching instructions into the window: If

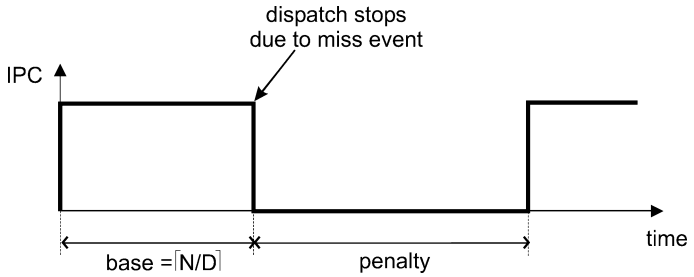

Fig. 3. Behavior of an interval in isolation.

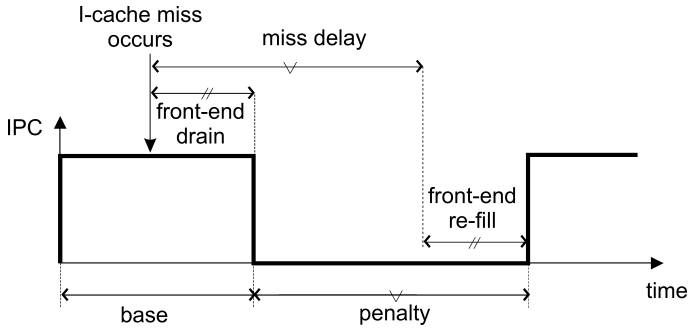

Fig. 4. An I-cache miss interval.

there are N instructions in a given interval (interval length of N) then it will take $\lceil N/D \rceil$ cycles to dispatch them into the window. The second part of the interval is the penalty part and is dependent on the type of miss event. The exact mechanisms which cause the processor to stop and restart dispatching instructions into the window, and the timing with respect to the occurrence of the miss event are dependent on the type of miss event, so each type of miss event must be analyzed separately, which we do now in the following subsections.

3.1.1 *I-Cache and I-TLB Misses.* L1 I-cache misses, L2 instruction misses, and I-TLB misses are the easiest cases to handle; they are described together because their behavior is similar. For simplicity, the discussion will be in terms of the I-cache miss case. Refer to Figure 4. At the beginning of the interval, instructions begin to fill the window at a rate equal to the maximum dispatch width. Then, at some point, an instruction cache miss occurs. Fetching stops, but the instructions already in the front-end pipeline are dispatched into the window (the pipeline drains), and then dispatch stops. The front-end pipeline drain takes a number of clock cycles equal to the number of front-end pipeline stages. After a delay for handling the I-cache miss, the pipeline begins to refill and dispatch is resumed. The front-end pipeline refill time is the same as the drain time; they offset each other. Hence, the overall dispatch stall time, that is, the penalty for an I-cache miss, is the miss delay. In summary, the time for an interval ending with an L1 I-cache miss equals $\lceil N/D \rceil + c_{iL1}$, with c_{iL1} the

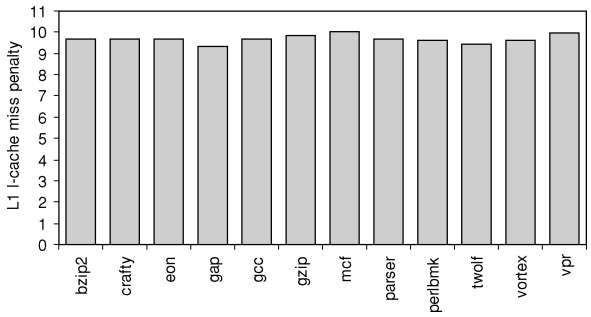

Fig. 5. The penalty due to an L1 instruction cache miss; the access latency for the L2 cache is 10 cycles.

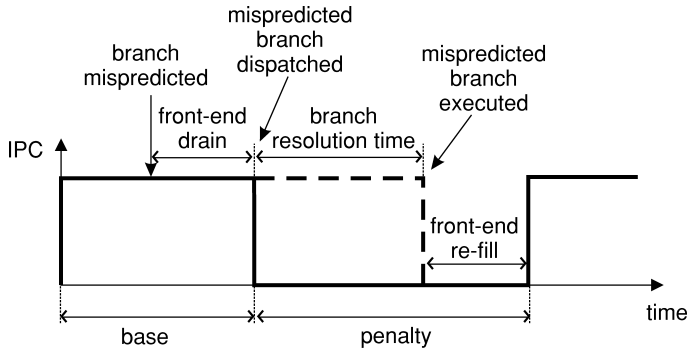

Fig. 6. Interval behavior for a branch misprediction.

miss delay for the L1 I-cache miss. For an L2 miss it is $\lceil N/D \rceil + c_{iL2}$; for an ITLB miss it is $\lceil N/D \rceil + c_{iTLB}$.

Note that the presence of the fetch buffer (Figure 1) has relatively little effect on this performance behavior. The fetch buffer is only present for compensating for cache line boundary effects.

Simulation data verifies the interval model for L1 I-cache misses; see Figure 5. Unless otherwise stated, these and other preliminary simulation results verifying model components use a baseline four-wide processor as given in Table III. In this experiment only L1 I-caches misses are allowed to occur (all other caches and the branch predictor are ideal), and we assume an L2 access latency of 10 cycles. The observed L1 I-cache miss penalty is relatively constant across all benchmarks. The slight fluctuation of the I-cache miss penalty between 9 and 10 cycles is due to the presence of the fetch buffer in the simulated instruction delivery subsystem. We obtained similar results for the L2 I-cache and I-TLB.

3.1.2 *Branch Mispredictions.* Ostensibly, behavior of branch mispredictions would appear to be similar to I-cache misses, but they are not. Figure 6 shows the timing for a branch misprediction interval. At the beginning of the interval, instructions are dispatched into the window, until, at some point, the mispredicted branch enters the window. Although wrong-path instructions continue to be dispatched (as displayed with the dashed line in Figure 6), from a

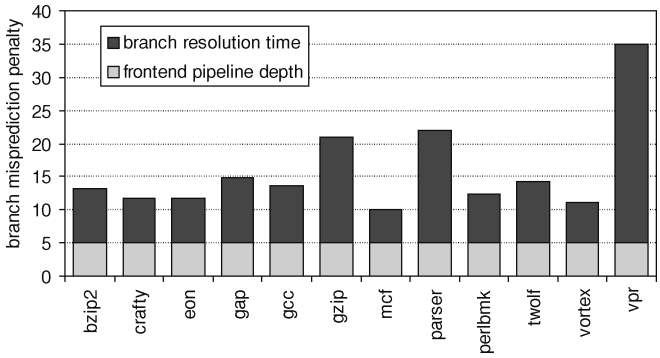

Fig. 7. The average penalty per mispredicted branch.

modeling perspective, dispatch of useful instructions stops at that point. Then, useful dispatch does not resume until the mispredicted branch is resolved, the pipeline is flushed, and the instruction front-end pipeline is refilled with correct-path instructions.

The overall performance penalty due to a branch misprediction thus equals the difference between the time the mispredicted branch enters the window and the time the first correct-path instruction enters the window following discovery of the misprediction. In other words, the overall performance penalty equals the branch resolution time, that is, the time between the mispredicted branch entering the window and the branch being resolved, plus the front-end pipeline length c_{fe}. In prior work (see Eyerman et al. [2006b]) we found that the mispredicted branch very often is the last useful instruction to be executed, and as such, the branch resolution time can be approximated by the window drain time c_{dr}, which we do in our model. As such, the total execution time for a branch misprediction interval equals $\lceil N/D \rceil + c_{dr} + c_{fe}$.

For many programs, the branch resolution time is the main contributor to the overall branch misprediction penalty (not the pipeline refill time). And this branch resolution time is a function of the dependence structure of the instructions in the window; that is, the longer the dependence chain and the execution latency of the instructions leading to the mispredicted branch, the longer the branch resolution time. The importance of the branch resolution time is illustrated in Figure 7 which decomposes the overall branch misprediction penalty in its two components, the branch resolution time and the front-end pipeline refill time, which is five cycles in this experiment. The branch resolution time is program dependent and varies between 5 and 30 cycles; the branch resolution time is subject to the interval length and the amount of ILP in the program; that is, the longer the interval and the lower the ILP, the longer the branch resolution time takes [Eyerman et al. 2006b]. For example, the branch resolution time for vpr is very large: This reflects a long dependence chain towards the mispredicted branch.

3.1.3 *Short Back-End Miss Events.* An L1 D-cache miss is considered to be a "short" back-end miss event, and is modeled as if it is an instruction that is

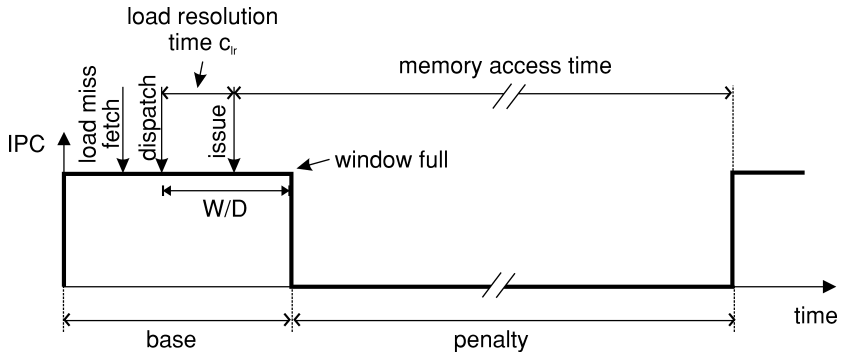

Fig. 8. Interval behavior for an isolated long back-end miss event.

serviced by a long-latency functional unit, the same as a multiply or a divide. In other words, it is assumed that the miss latency can be hidden by out-of-order execution, and this assumption is incorporated into our definition of a balanced processor design. In particular, the ILP model for balanced processor designs that we will present in Section 3.2 includes L1 D-cache miss latencies as part of the average instruction latency when balancing ROB size and issue width under ideal conditions.

3.1.4 *Long Back-End Miss Events.* When a long data cache miss occurs, that is, from the L2 cache to main memory, the memory delay is typically quite large: on the order of hundreds of cycles. Similar behavior is observed for D-TLB misses. Hence, both are handled in a similar manner.

On an isolated long data cache miss, the ROB fills because the load blocks the ROB head, then dispatch stalls, and eventually issue and commit cease [Karkhanis and Smith 2002]; see Figure 8. After the miss data returns from memory, the ROB becomes unblocked and instruction dispatch resumes. The total long data cache miss penalty equals the time between the ROB fill and the time data returns from memory. The penalty for an isolated long back-end miss thus equals the main memory access latency minus the number of cycles where useful instructions are dispatched under the long-latency miss. These useful instructions are dispatched between the time the long-latency load dispatches and the time the ROB blocks after the long-latency load reaches its head; this is the time it takes to fill the entire ROB, W/D, minus the time it takes for the load to issue after it has been dispatched, or the load's resolution time, c_{lr}. As such, the execution time for an isolated long back-end miss interval equals $\lceil N/D \rceil + c_{L2} - (W/D - c_{lr})$.

For multiple long back-end misses that are independent of each other and that occur within an interval of W instructions (the ROB size), the penalties overlap completely [Chou et al. 2004; Karkhanis and Smith 2004, 2002]. This is illustrated in Figure 9 where there are $S < W$ instructions separating two long-latency misses. After the first load receives its data and unblocks the ROB, S more instructions dispatch before the ROB blocks for the second load, and the time to do so, S/D, offsets an equal amount of the second load's miss penalty.

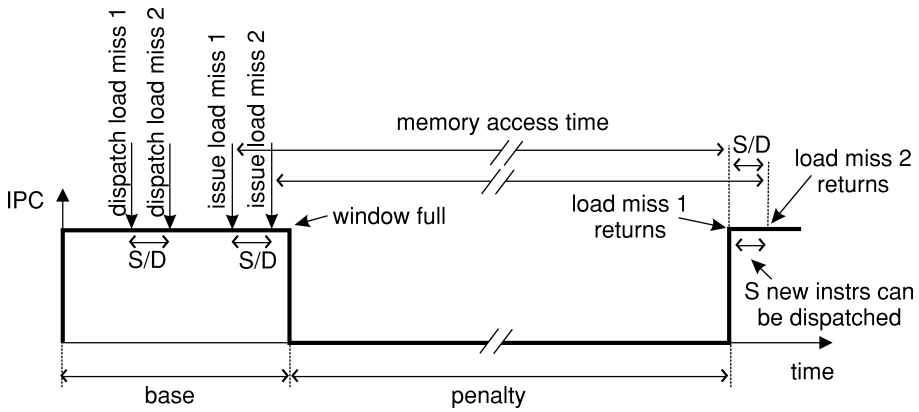

Fig. 9. Interval behavior for overlapping long back-end miss events.

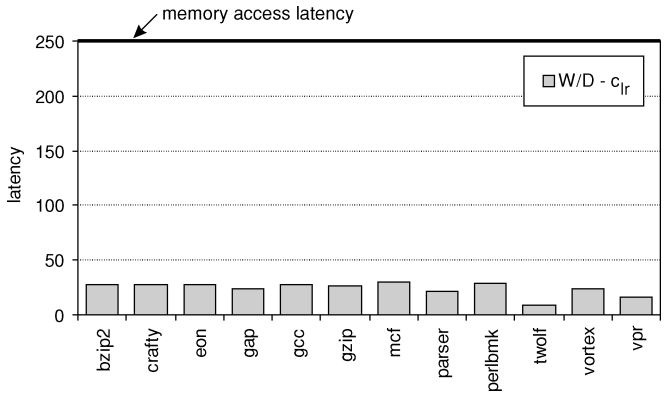

Fig. 10. Quantifying the $W/D - c_{lr}$ term in comparison to the main memory access time which is assumed to be 250 cycles; $W = 128$ and $D = 4$ in our baseline processor model; see Section 3.5.

This generalizes to any number of overlapping misses, so the penalty for a burst of independent long-latency back-end misses equals the penalty for an isolated long-latency load, namely $c_{L2} - (W/D - c_{lr})$. Because the amount of useful work done under the long-latency loads, $W/D - c_{lr}$, is relatively small compared to the main memory access latency c_{L2} (see the simulation results presented in Figure 10), we assume this term is zero and approximate the penalty for isolated and overlapping long-latency loads as c_{L2}.

Figure 11 shows simulation results that quantify the average penalty per long-latency data cache miss assuming a 250-cycle main memory access time. The two bars represent the average penalty for nonoverlapping long-latency data cache misses, and for all (both overlapping and nonoverlapping) long-latency data cache misses, respectively. The average penalty per long-latency miss is significantly smaller than 250 cycles because of overlapping misses for most of the programs; that is, Memory-Level Parallelism (MLP) [Glew 1998; Chou et al. 2004] partially hides main memory access time, the most notable examples being crafty and gzip. For some programs, however, there is little or

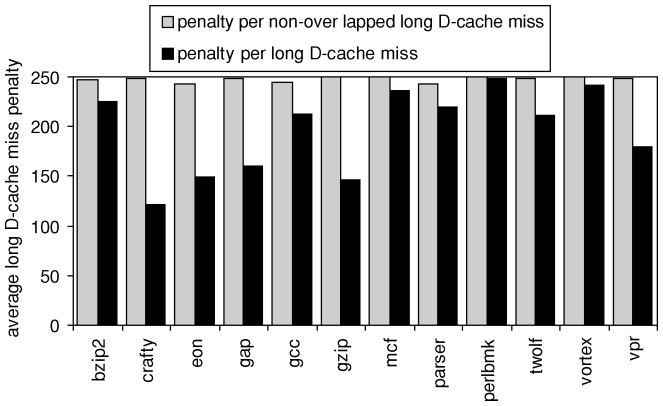

Fig. 11. The average penalty per nonoverlapping long-latency L2 data cache miss (the number of cycles lost due to L2 data cache misses divided by the number of nonoverlapping long-latency misses) versus the average penalty per long-latency L2 data cache miss (the number of cycles lost due to L2 data cache misses divided by the total number of long-latency misses, both overlapping and nonoverlapping misses); main memory access time is assumed to be 250 cycles.

no memory-level parallelism to be exploited although there is a large number of long-latency data cache misses, as is the case for mcf: The long-latency misses are dependent upon each other due to pointer chasing code which serializes the long-latency misses.

3.1.5 *Miss Event Overlaps.* Thus far, we have modeled the various miss event types in isolation. However, the miss events may interact with each other. The interaction between front-end miss events (branch mispredictions and I-cache misses) is limited because they serially disrupt the flow of instructions and their penalties do not overlap. Long-latency back-end miss events interact frequently and have a large impact on overall, performance, as discussed in the previous section, but these can be modeled fairly easily by counting the number of independent long-latency back-end misses that occur within an instruction sequence less than or equal to the window size W. The interactions between front-end miss events and long-latency back-end miss events are more complex because front-end miss events can overlap back-end misses; however, these interactions are relatively minor, as discussed in the following paragraphs.

We first consider a mispredicted branch following a long D-cache miss. If the mispredicted branch is not yet in the ROB when the long miss blocks dispatch, then the branch penalty and the long D-cache miss penalty will serialize. If the mispredicted branch does make it into the ROB before dispatch blocks, then there are two possible scenarios. In one scenario the long D-cache miss feeds the mispredicted branch, that is, the load miss reads data on which the branch depends, and, as a result, the miss penalties serialize. In the other scenario where the mispredicted branch does not depend on the long D-cache miss, the branch misprediction penalty is hidden under the long D-cache miss penalty.

Table I. Percentage Cycles for which Front-End Miss Penalties Overlap with Long Back-End Miss Penalties

benchmark	input	% overlap
bzip2	program	0.12%
crafty	ref	1.03%
eon	rushmeier	0.01%
gap	ref	5.40%
gcc	166	0.93%
gzip	graphic	0.04%
mcf	ref	0.02%
parser	ref	0.43%
perlbmk	makerand	1.00%
twolf	ref	4.97%
vortex	ref2	3.10%
vpr	route	0.89%

If a short I-cache miss happens while a long D-cache miss is outstanding, then the miss handling is completely overlapped with the long D-cache miss and does not affect performance. If there is a long I-cache miss or an I-TLB miss that occurs after the long D-cache miss, then both misses essentially overlap. In particular, if the I-cache miss instruction enters the ROB before the load issues, then the I-cache miss is hidden under the D-cache miss. In the other case where the I-cache miss instruction enters the ROB after the load issues, then the D-cache miss is hidden under the I-cache miss.

By studying simulation results we observe that the number of overlaps between front-end and long-latency back-end miss events tend to be rare. Table I reports the fraction of the total cycle count for which overlaps are observed between front-end miss event penalties (L1 and L2 I-cache misses, I-TLB misses, and branch mispredictions) and long-latency back-end miss event penalties (L2 D-cache misses and D-TLB misses). Overlaps are no more than 1% for most benchmarks, and only as much as 5% for a couple of benchmarks.

Accounting for all the overlap effects between front-end and back-end miss events complicates the model, although it is possible to do. In order to compute interval lengths in the presence of overlaps, one would need to collect miss event data (see Section 3.3.3) for all combinations of caches and predictors. This would also introduce a number of complicating terms in the performance model, which, in most cases, would be insignificant. We therefore ignore these overlap effects in the development of the model.

3.2 Modeling Instruction-Level Parallelism

An important element of out-of-order processor performance is the inherent ILP in the program being executed. We characterize the amount of ILP in a program by measuring the average critical path length over windows of W consecutive dynamic instructions [Michaud et al. 2001]; see Figure 12. Conceptually, the window slides along the dynamic instruction stream, and the critical path is the longest data dependence chain for that window. Intuitively, the window cannot slide any faster than the processor can issue the instructions belonging to the critical path.

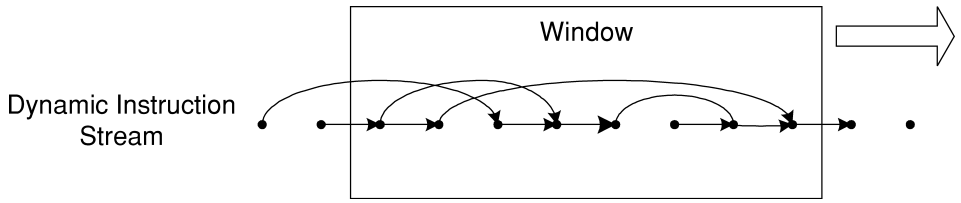

Fig. 12. The dynamic instruction stream with data dependence arcs added. As the window slides over the dynamic instruction trace, it can move no faster than it can consume dependent instructions along the critical path.

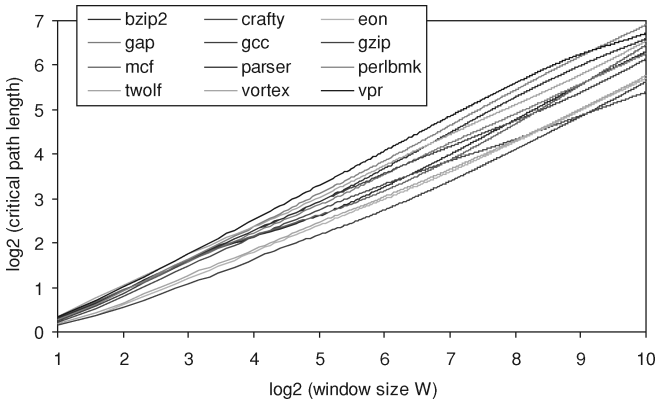

Fig. 13. The average critical path length $K(W)$ as a function of window size on a log-log scale.

For a given program, we define $K(W)$ to be the average critical path length for a window of size W. For the SPEC CPU2000 integer benchmarks, we have evaluated $K(W)$, and the results are plotted in Figure 13 on a log-log scale. As was observed by Michaud et al. [2001], these curves follow an approximate straight line (at least in the region of practical out-of-order processors). Hence, there is a power law relationship between W and $K(W)$; namely

$$K(W) = \frac{1}{\alpha} \cdot W^{1/\beta}, \beta \geq 1, \tag{1}$$

in which the values of α and β are program dependent. The higher the value of β, the shorter the critical path and the more ILP is present; hence β is a measure of the inherent ILP in a program. Table II contains α and β values for benchmark programs. We observe that β takes on a range of values from 1.31 to 1.79. In prior work [Michaud et al. 2001, 1999], it is assumed that $\beta \approx 2$.

Note that an algorithm of complexity $O(N \cdot W_{max})$ can be used for computing $K(W)$ with N being the number of instructions in the dynamic instruction stream and W_{max} the maximum window size that one is interested in. This algorithm exploits the fact that the critical path in a window of size W is longer than the critical path in a window of size V, with $W > V$.

Assuming all instructions have unit latency, and applying Little's law, the average rate (instructions per cycle) at which the window can pass over the

Table II. Power Law Estimate of $K(W)$ as a Function of α and β

	vpr	perlbmk	parser	twolf	gap	bzip2	gzip	mcf	eon	vortex	gcc	crafty
α	1.40	1.55	1.59	1.26	1.34	1.46	1.16	1.35	1.51	1.44	1.67	1.06
β	1.31	1.33	1.36	1.47	1.50	1.53	1.60	1.61	1.66	1.67	1.68	1.79

Benchmarks are sorted by increasing β.

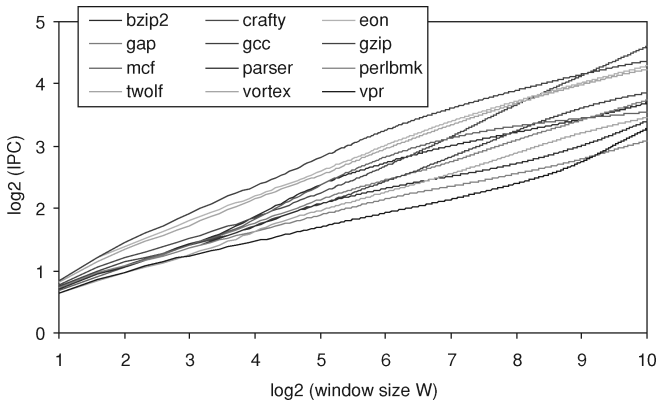

Fig. 14. The IW characteristic on a log-log scale.

entire instruction sequence for a given program is

$$IPC = \frac{W}{K(W)} = \alpha \cdot W^{1-1/\beta}, \tag{2}$$

which we call the *IW characteristic*, displayed in Figure 14. This is the average effective instruction throughput (on the vertical axis) as a function of window size (on the horizontal axis). Evaluating the IW characteristic for unit-latency instructions yields an ILP measure that is independent of processor implementation characteristics. For a given implementation, if the average instruction latency is ℓ (comprising nonunit instruction execution latencies and short L1 data cache misses), then the critical path length is increased by a factor of ℓ, so

$$IPC = \frac{W}{\ell \cdot K(W)} = \alpha \cdot \frac{W^{1-1/\beta}}{\ell}. \tag{3}$$

If we use the approximation $\beta \approx 2$, then

$$IPC = \alpha \cdot \frac{\sqrt{W}}{\ell}; \tag{4}$$

this reflects a square root relationship between the window size and instruction throughput [Riseman and Foster 1972; Wall 1991; Michaud et al. 1999].

Note that the straight-line behavior in the critical path graph (Figure 13) extends out to windows (ROB sizes) at least as large as 1K instructions. The corresponding IPC (using the equation just given) is eight or more; see Figure 14. In the worst cases, for example, in vpr and perlbmk, a window size of 1K entries is needed for achieving an IPC of eight. The reason is that these programs are dominated by very long critical paths (due to loop-carried dependences) and

there are not enough instructions off the critical path for sustaining a wide effective issue rate. For most of the other benchmarks, however, an IPC of eight can be achieved with a significantly smaller ROB.

In some circumstances, as we will see later in this article, one may be interested in the inverse relation between instruction throughput and window size.

$$W = \left(\ell \cdot \frac{IPC}{\alpha}\right)^{\beta/(\beta-1)} \tag{5}$$

This equation indicates the required ROB size W for attaining a given IPC.

3.3 Interval Model

In the previous sections, we presented equations for the various types of miss event intervals. We now construct the overall model by summing over the miss event types.

3.3.1 *Dispatch Inefficiency.* First, we consider the $\lceil N/D \rceil$ terms that are part of every miss interval. If we temporarily ignore the ceiling function performed over each interval, these terms sum to N_{total}/D, with N_{total} being the total dynamic instruction count. The ceiling function results from an additional "edge effect" delay whenever an interval length N is not an integer multiple of the processor dispatch width D. If we assume that the residues for the intervals are evenly distributed over the D possibilities $0, 1, \ldots, D-1$, then we can compute the average $\lceil N/D \rceil$ as $N/D+(D-1)/2D$. We denote the miss event counts[2] for L1 I-cache misses, L2 I-cache misses, branch mispredictions, nonoverlapping long data cache (L2) misses as $m_{iL1}, m_{iL2}, m_{br}, m^*_{dL2}(W)$, respectively.[3] The total number of intervals then equals $m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)$. Hence the total time in clock cycles to dispatch N_{total} instructions on a processor with width D accounting for miss event edge effects equals

$$\sum \lceil N/D \rceil \approx \frac{N_{total}}{D} + \frac{D-1}{2D} \cdot (m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)). \tag{6}$$

The second term in the previous formula (accounting for miss event edge effects) can be considered as an *inherent dispatch inefficiency*. It is caused by the mere presence of miss events, and the only way to remove it is to eliminate the miss events. This is in contrast to implementation inefficiencies such as fetch units that do not fetch beyond taken branches, for example. It follows that for shorter intervals, the inherent dispatch inefficiency is relatively larger.

To evaluate this effect we simulated benchmarks with a processor having dispatch width 4. We found that for most benchmarks interval edge effects result in an achieved dispatch rate of between 3.8 and 4. For the benchmark mcf with the most miss events (one per 13 instructions on average) the effective dispatch width is 3.6 instructions per cycle.

[2]We do not include TLB misses here in the model to simplify the readability of the formulas; their modeling is similar to the modeling of long cache misses.

[3]We denote the number of nonoverlapping long D-cache misses with an asterisk *, and indicate that the number of nonoverlapping long D-cache misses is dependent on the processor's window size W.

3.3.2 *Putting It Together: The Overall Model.* Now that the $\lceil N/D \rceil$ terms are accounted for, the rest of the miss interval times are $m_{iL1} \cdot c_{iL1}$ for the L1 I-cache misses, $m_{iL2} \cdot c_{L2}$ for the L2 I-cache misses, $m_{br} \cdot (c_{dr} + c_{fe})$ for the branch mispredictions, and $m^*_{dL2}(W) \cdot c_{L2}$ for the nonoverlapping L2 D-cache misses.

The total execution time, measured in cycles, C is then

$$\begin{aligned} C = \frac{N_{total}}{D} + \frac{D-1}{2D} \cdot (m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)) + \\ m_{iL1} \cdot c_{iL1} + m_{iL2} \cdot c_{L2} + \\ m_{br} \cdot (c_{dr} + c_{fe}) + \\ m^*_{dL2}(W) \cdot c_{L2}. \end{aligned} \tag{7}$$

3.3.3 *Providing Input Parameters for the Model.* To provide data values for the model parameters, we rely on two general types of program characteristics. The first set of program characteristics are related to a program's locality behavior and include the miss rates and interval lengths for the various miss events. The second program characteristic relates to the window drain time c_{dr}. Note that these two sets of program characteristics are the only program characteristics needed by the model; all the other parameters are microarchitecture-related. Although a number of ways could be used to arrive at the program-related model parameters, we now discuss specific methods that yield an accurate model.

—*Characterizing Locality Behavior*. Caches and predictors exhibit behavior based on temporal and/or spatial locality. Modeling cache and branch predictor locality behavior and performance is an important research topic of its own, and a significant amount of research has been directed at caches in particular; see for example Zhong et al. [2003], Berg and Hagersten [2005], and Guo and Solihin [2006]. To account for cache and predictor behavior in our model, we could incorporate such models. However, we prefer to keep the modeling of caches and predictors (and the errors they introduce) separate from the rest of the out-of-order processor model.

Consequently, we incorporate a trace-driven approach for determining cache and branch predictor miss rates and other locality behavior. This can be done fairly efficiently via the time-honored method of functionally simulating the program to generate a program trace, and then feeding the trace into a cache or branch predictor analyzer. Such an analyzer can simultaneously generate miss rate statistics for a number of configurations. Note that for this approach a detailed out-of-order processor simulator is not required. Only a functional simulator and specialized trace-driven cache and branch predictor analyzers are required. And, if available, results from a specialized cache or predictor model could be used as well.

Besides miss rates, we also determine interval lengths, namely the number of instructions in the dynamic instruction stream between two miss events. In particular, we compute the branch misprediction interval length distribution $P_{br}[N = n]$. We also compute the number of *nonoverlapping* independent long-latency data cache misses $m^*_{dL2}(W)$ within an arbitrary window size W. To

do so, we use an algorithm of complexity $O(N \cdot W_{max})$, with N the number of instructions in the dynamic instruction stream and W_{max} the maximum window size one is interested in. This algorithm is incorporated in the trace-driven cache simulator and scans the instruction trace only once. This algorithm is similar to the one used for generating the critical path curve $K(W)$; we exploit the fact that the number of nonoverlapping misses in a window of size W is larger than the number of nonoverlapping misses in a window of size V, with $W > V$.

—*Estimating Window Drain Time*. The second program characteristic that serves as input to the interval model is the average window drain time, c_{dr}, following a branch misprediction.

There are two main considerations when estimating window (ROB) drain time. First, we need to know the number of useful instructions in the window at the time it begins to drain, and second, we need to know the drain rate in order to derive the drain time. Both are related to: (i) the interval length, (ii) the ILP in the program being executed, and (iii) the instruction execution latencies. The shorter the interval, the fewer instructions in the window when the mispredicted branch enters the window, and thus the shorter the window drain time. Also, the lower the ILP, the slower the window will drain because of long data dependence paths. Similarly, the longer the average instruction execution latency, the longer it takes to drain the window.

The window drain time $c_{dr}(n)$ for a given interval length n is computed using the ILP model discussed in Section 3.2. This is done in two steps. We first compute the number of instructions residing in the window at the time the mispredicted branch enters. This is done by evaluating the following recursive equations, assuming $I_0 = n, w_0 = 0$.

$$\begin{cases} I_i = I_{i-1} - D \\ w_i = w_{i-1} + D - \frac{w_{i-1}}{\ell \cdot K(w_{i-1})} \end{cases} \tag{8}$$

Initially, the window is empty ($w_0 = 0$) and n instructions need to be dispatched ($I_0 = n$). The equations state that D instructions are placed into the window each cycle, and $w_{i-1}/(\ell \cdot K(w_{i-1}))$ instructions are removed (following Eq. (3)). The recurrence evaluation terminates at step (cycle) j if $I_j = 0$. The number of instructions remaining in the window then equals w_j. In a second step, we compute the window drain time; this is done using the $K(W)$ measure: $c_{dr}(n) = \ell \cdot K(w_j)$, that is, it takes $\ell \cdot K(w_j)$ cycles to drain w_j instructions.

Once the window drain time $c_{dr}(n)$ for an interval length n is known, the average window drain time c_{dr} can be computed by multiplying the probability $P_{br}[N = n]$ for an interval of length n with the window drain time for an interval of length n: $c_{dr} = \sum_{n=1}^{\infty} P_{br}[N = n] \cdot c_{dr}(n)$.

3.4 Comparison to Other Models

A number of researchers have considered superscalar out-of-order processor models [Dubey et al. 1994; Noonburg and Shen 1997, 1994; Sorin et al. 1998; Michaud et al. 2001, 1999; Eeckhout and De Bosschere 2001; Hartstein and

Puzak 2002; Fields et al. 2004; Karkhanis and Smith 2004; Ipek et al. 2006; Lee and Brooks 2006; Joseph et al. 2006a, 2006b; Taha and Wills 2008], but four primary efforts led us to the interval model as described here. First, Michaud et al. [2001] focus on performance aspects of instruction delivery and model the instruction window and issue mechanisms. They show a power law (roughly a square law) relationship between window size and issue rate. This is in line with much earlier work dating back to Riseman and Foster [1972] and Wall [1991]. Second, Karkhanis and Smith [2004] extend this type of analysis to all types of miss events and build a complete performance model, which includes a sustained steady-state performance rate punctuated with gaps that occur due to miss events. Third, Taha and Wills [2003] also break instruction processing into intervals (which they call "macro blocks"); see also their follow-on work [Taha and Wills 2008]. The macro block execution times are estimated using empirical average characteristics obtained from extensive detailed cycle-level microarchitecture simulations. Although developed independently, the equations in the mechanistic interval model have a similar structure to those in the model of Hartstein and Puzak [2002]. Because the organization of the Hartstein-Puzak equations have intuitive appeal, we organize the components of the interval model in a similar manner; see Section 4.2.1.

The interval model developed in this article models performance in terms of the processor's dispatch behavior, which is in contrast to prior work in mechanistic processor modeling which models performance in terms of the processor's issue rate; see for example Michaud et al. [1999], Karkhanis and Smith [2004], and Taha and Wills [2008]. This significantly simplifies the model and makes it more intuitive. In addition, the interval model presented in this article extends prior analytical out-of-order processor modeling work in two major ways. First, interval analysis exposes the impact of the interval behavior on overall performance. This is reflected in how the interval model deals with: (i) the dispatch inefficiency (more miss events and thus more intervals increase the dispatch inefficiency) and (ii) the branch resolution time (longer interval lengths result in longer branch resolution times). Second, our mechanistic model estimates the branch resolution time using the average critical path length $K(W)$ that characterizes a program's inherent ILP. No extensive detailed processor simulations are needed as done in Karkhanis and Smith [2004] and Taha and Wills [2008] for computing the IW characteristic.

3.5 Model Validation

Before studying resource scaling in superscalar out-of-order processors, we demonstrate the model's accuracy using the SPEC CPU2000 integer benchmarks (assuming SimPoint's single simulation points [Sherwood et al. 2002]) and the processor configurations tabulated in Table III (the 4-wide processor will serve as our baseline processor configuration); the caches and branch predictor are smaller than what is typically observed in contemporary superscalar processors in order to stress the model. The simulator we are validating our model against is the SimpleScalar out-of-order processor simulator [Burger and Austin 1997]. Figure 15 shows the model's overall predicted

Table III. The Baseline Processor Assumed in Our Experimental Setup

2-wide processor	
processor width	$D = R = I = 2wide$
ROB	64 entries
4-wide processor (baseline)	
processor width	$D = R = I = 4$ wide
ROB	128 entries
6-wide processor	
processor width	$D = R = I = 6$ wide
ROB	256 entries
8-wide processor	
processor width	$D = R = I = 8$ wide
ROB	512 entries
other configuration parameters	
fetch width F	$F = 2 \cdot D$
fetch buffer	F entries
latencies	load 2 cycles, mul 3 cycles, div 20 cycles, arith/log 1 cycle
L1 I-cache	8KB direct-mapped, 32-byte cache lines
L1 D-cache	16KB 4-way set-associative, 32-byte cache lines
L2 cache	unified, 1MB 8-way set-associative, 128-byte cache lines 10 cycle access time
main memory	250 cycle access time
branch predictor	hybrid predictor consisting of 4K-entry meta, bimodal and gshare predictors
front-end pipeline	5 stages

IPC compared to simulation for 2-wide, 4-wide, 6-wide, and 8-wide processor configurations.

We observe that the interval model tracks the performance differences fairly well across benchmarks and processor configurations. The average IPC difference with respect to simulation for the 4-wide processor is 7%; the average IPC differences for the 2-wide, 6-wide, and 8-wide machines are 3.9%, 9.9%, and 13.1%, respectively. The largest differences are observed for parser, gzip, vpr, and perlbmk with 21.4%, 16.9%, 13.8%, and 9.1% IPC prediction errors for the baseline configuration, respectively.

To evaluate the error due to individual model components, it is simpler to use Cycles Per Instructions (CPI), the reciprocal of IPC. Dividing Eq. (7) by the number of instructions executed N_{total} will yield the overall CPI, which is composed of individual CPI components (also called "CPI adders" [Emma 1997; Brooks et al. 2000]) consisting of a base CPI component plus a number of CPI components that reflect "lost" cycle opportunities due to miss events such as branch mispredictions and cache misses. Figure 16 breaks down overall CPI into its major CPI components, for the baseline processor configuration.

For gzip and perlbmk, the main error is in estimating the branch misprediction penalty. For parser and vpr, the main error is in estimating the number of overlapping long-latency loads. More specifically, when a long-latency load occurs after a mispredicted branch that is dependent on a preceding long-latency load, the model assumes that both long-latency loads overlap, however, they actually serialize. This inaccuracy could be modeled by cosimulating the cache

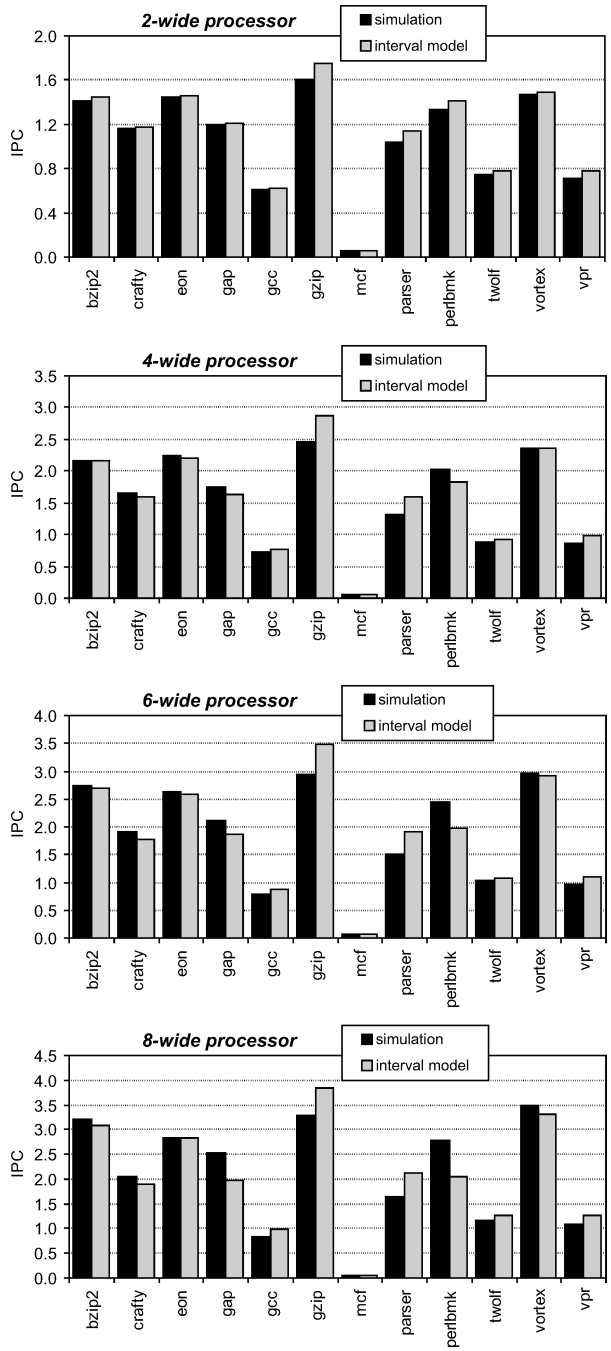

Fig. 15. Difference in IPC predicted by the interval model versus simulation.

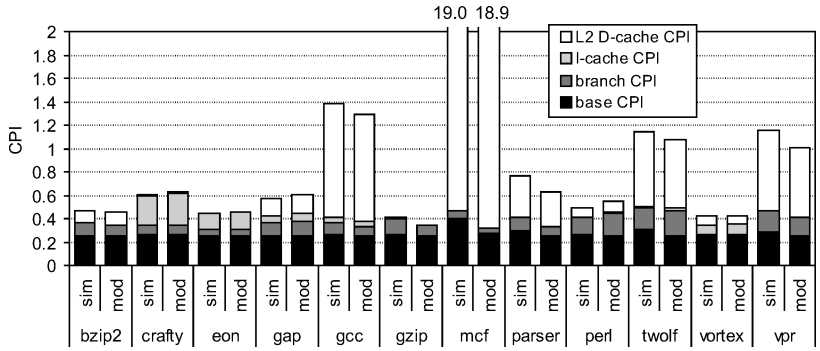

Fig. 16. Comparing the CPI components predicted by the interval model ("mod") versus simulation ("sim") for the baseline 4-wide out-of-order processor configuration.

and branch predictor, however, this may be impractical when multiple combinations of cache and branch predictor configurations are to be explored, as mentioned earlier. The error in the base CPI component for mcf (see Figure 16) comes from the assumption that L1 D-cache misses are hidden by out-of-order execution in a balanced processor; however, mcf has a large number of L1 D-cache misses which extends the critical path to the point where it can no longer be hidden by out-of-order execution.

4. RESOURCE SCALING IN OUT-OF-ORDER PROCESSORS

To demonstrate the fundamental insight that the interval model provides, we now use the model for studying resource scaling in out-of-order processors. In prior work, we used interval analysis for a number of other applications (we refer the interested reader to the respective publications: studying the branch misprediction penalty [Eyerman et al. 2006b], building a hardware performance counter architecture for accurate CPI component construction [Eyerman et al. 2007, 2006a], and driving the automated design of application-specific superscalar out-of-order processors [Karkhanis and Smith 2007]).

In this section, we focus on: (i) resource scaling in balanced processor designs, (ii) optimal pipeline depth and width scaling and the relationship between optimum depth and width, and (iii) potential benefits of overprovisioned processor designs.

4.1 Balanced Processor Design

Recall that we define a processor design to be balanced if there are adequate resources to sustain an instruction throughput of D instructions per cycle in the absence of miss events. Furthermore, the resources should not exceed those that are adequate; that is, reducing the ROB size (and/or related structures) of a balanced design results in significant performance degradation. As pointed out in prior work [Riseman and Foster 1972; Wall 1991, Michaud et al. 2001, 1991] and discussed in Section 2, for most practical processor widths, programs contain enough ILP that balance can be achieved.

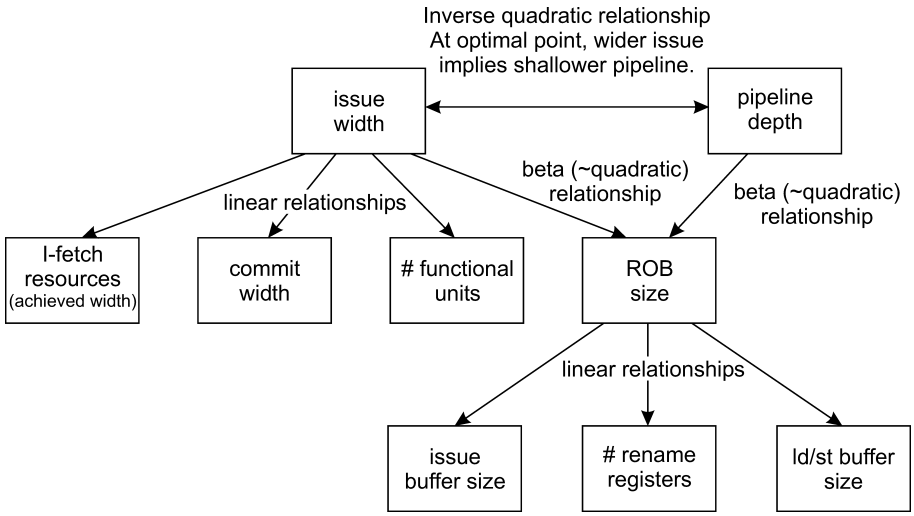

Fig. 17. Structure scaling graph illustrating the relationships between various processor structures in a balanced processor design.

The equation $W = (\ell \cdot IPC/\alpha)^{\beta/(\beta-1)}$ as derived in Section 3.2 expresses the relationship between the ROB size W and the instruction throughput IPC. This formula essentially characterizes the way that the ROB size should scale with respect to the processor width D and pipeline depth for balanced processor designs. Increasing the dispatch width D by a factor S_w implies that the ROB size should scale by a factor $S_w^{\beta/(\beta-1)}$. Similarly, increasing pipeline depth by a factor of S_d increases the instruction execution latencies ℓ by a factor S_d and thus implies that the ROB size should scale by a factor $S_d^{\beta/(\beta-1)}$. In other words, for a balanced processor design, ROB size scales superlinearly with both pipeline width and depth because $\beta/(\beta-1) > 1$. Prior work, and results reported in Section 3.2 for the SPEC CPU2000 integer benchmarks, indicate ROB size scales at least quadratically with D. In the following discussion, we will refer to this as a "quadratic" relationship (recognizing that it may be more than quadratic).

These observations along with a straightforward analysis of how microarchitecture structures are related to dispatch width and ROB size leads to the *scaling graph* shown in Figure 17. This scaling graph can be used to guide the sizing of the various hardware structures: For a given pipeline width and depth, the other microarchitecture structures follow the relationships shown in Figure 17 for achieving a balanced design. The ROB size shows a quadratic "beta" relationship ($S^{\beta/(\beta-1)}$) with both the pipeline width D and pipeline depth. The other structures show a linear relationship with either the pipeline width or ROB size. For example, the I-fetch resources, commit width, and the number of functional units show a linear relationship with the dispatch width; issue buffer size, the number of physical registers, and load/store buffer sizes show a linear relationship with ROB size.

4.2 Pipeline Depth and Width

We now use the mechanistic model for studying pipeline configurations: depth, width, and their relationship. In this section, we assume balanced processor designs, that is, when we scale depth and width we also scale the processor structures (ROB size, issue buffer sizes, number of functional units, etc.) to maintain balance as described in the previous section.

4.2.1 *Pipeline Depth.* Although the optimum pipeline depth is a well-researched topic Kunkel and Smith 1986; Emma and Davidson 1987; Dubey and Flynn 1990; Agarwal et al. 2000; Hrishikesh et al. 2002; Sprangle and Carmean 2002; Srinivasan et al. 2002, Hartstein and Puzak 2003, 2002], the mechanistic model provides additional insights into ways that pipelining affects individual microarchitecture components, and in addition, provides the opportunity to study partial pipelining, which other models do not include.

We first extend the mechanistic model by converting from clock cycles to absolute time (say, in nanoseconds). To do so, we multiply the estimated cycle counts by the cycle time. We denote the cycle time as $t_S = t_o + t_p/p$, with t_o the latch overhead for a given technology, t_p the total logic (and wire) delay of a processor pipeline, and p the number of pipeline stages. The value p is essentially the pipeline depth. The classic five-stage RISC pipeline, where operations such as reading registers and an ALU operation require a single pipeline stage, has a value $p = 5$. Note that a realistic superscalar processor is composed of several interconnected pipelines, not a simple linear pipeline. However, if all the pipelines are designed with the same amount of logic and wires between stages, then t_p/p will be the same regardless of which pipeline is used for modeling.

Consequently, after organizing the terms along the lines of the model proposed by Hartstein and Puzak [2002], the total execution time then becomes

$$\begin{aligned} T = {} & \left(\frac{N_{total}}{D} + \frac{D-1}{2D} \cdot (m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)) \right) \cdot (t_o + t_p/p) + \\ & m_{iL1} \cdot c_{iL1} \cdot p/p_b \cdot (f_{iL1} \cdot t_o + t_p/p) + m_{iL2} \cdot c_{L2} \cdot p/p_b \cdot (f_{iL2} \cdot t_o + t_p/p) + \\ & m_{br} \cdot (c_{dr}(p, D) + c_{fe}) \cdot p/p_b \cdot (t_o + t_p/p) + \\ & m^*_{dL2}(W) \cdot c_{L2} \cdot p/p_b \cdot (f_{dL2} \cdot t_o + t_p/p). \end{aligned} \tag{9}$$

In this formula, p_b is the number of pipeline stages in a baseline configuration; the same holds for c_{iL1}, c_{L2}, and c_{fe}. The pipeline factors f ($0 \leq f \leq 1$) denote the degree to which the cache misses are pipelined; $f = 0$ denotes nonpipelined cache misses and $f = 1$ denotes fully pipelined cache misses; a cache miss is pipelined if the miss path logic contains pipeline latches and hence latch overhead. This is to account for the facts that cache misses do not occur with high frequency and RAMs lower in the memory hierarchy often take multiple cycles to access. Hence, although the miss path may be pipelined, it is often pipelined at an effectively slower clock rate than the processing pipeline. For example, if the L2 cache takes four cycles to access, the L1 miss path may be pipelined with a factor of $f = .25$. The advantage of reduced pipeline is somewhat lower latency. If all miss paths are fully pipelined (as is sometimes assumed), then

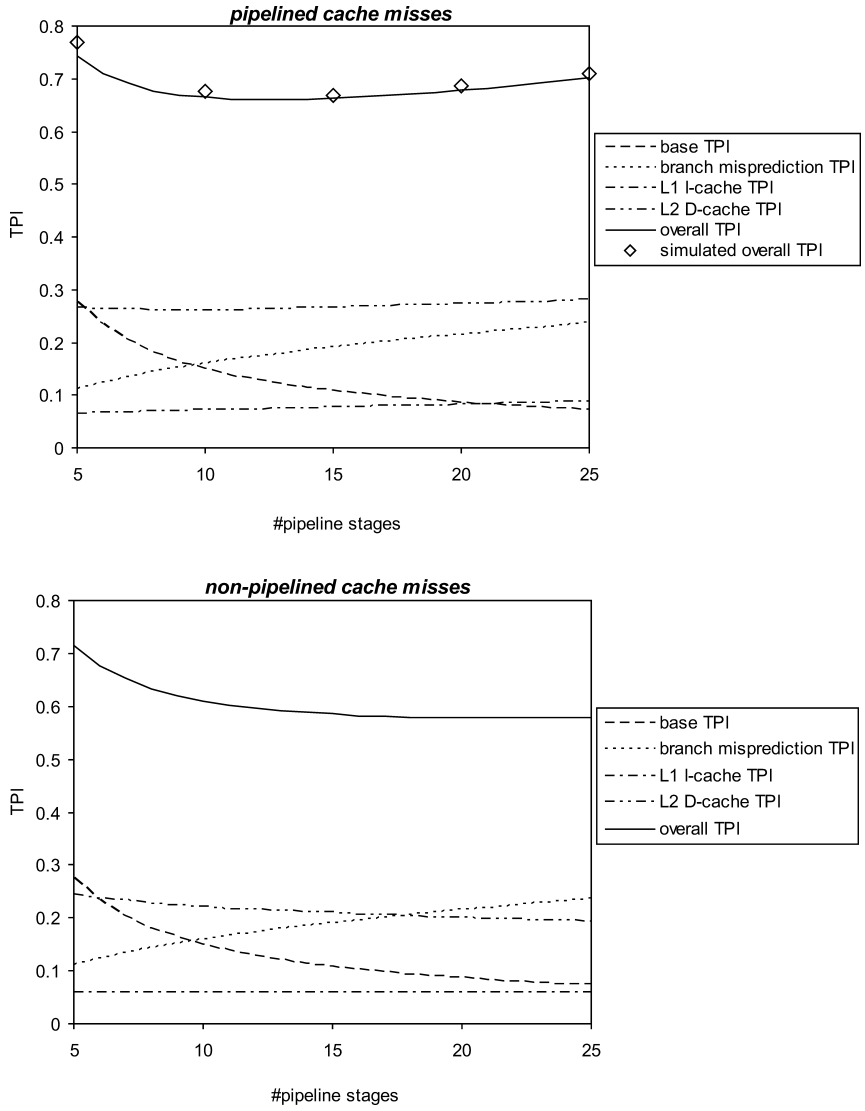

Fig. 18. TPI components and overall TPI as a function of depth, assuming pipelined cache misses (top graph) and nonpipelined cache misses (bottom graph).

the f terms can be removed to simplify the equations. Overall TPI (Time Per Instruction) is then computed by dividing the preceding total execution time T with the total dynamic instruction count N_{total}: $TPI = T/N_{total}$.

We now discuss how the four terms in the previous equation are affected by pipeline depth for balanced processor designs. We refer to Figure 18 which shows how the individual TPI components are affected by pipeline depth, averaged across the SPEC CPU2000 integer benchmarks. The top graph in Figure 18 assumes pipelined cache miss handling, whereas the bottom graph assumes nonpipelined cache miss handling. To further demonstrate the accuracy of the

mechanistic model as a function of pipeline depth, we also show the overall TPI numbers obtained through simulation in the top graph.

Base TPI. The first term (base TPI) is where the performance advantage of deeper pipelining appears. A higher clock frequency reduces the absolute time for performing useful work. The inherent dispatch penalty also decreases in absolute time with increased pipeline depth.

I-cache TPI. The I-cache TPI terms remain constant in terms of cycles (the p in the numerators and denominators cancel) but increase linearly in actual time because of pipelining overhead, in case of (partially or fully) pipelined cache misses; see top graph in Figure 18. In case of nonpipelined cache misses, the I-cache terms remain constant in absolute time; see bottom graph in Figure 18.

Branch misprediction TPI. The branch misprediction term can be broken up into two terms, one related to the branch resolution time (estimated by the model as the window drain time) and another one related to refilling the front-end pipeline. The pipeline refill term, $m_{br} \cdot c_{fe} \cdot p/p_b \cdot (t_o + t_p/p)$, increases with increasing pipeline depth because of pipeline overhead. The window drain term, $m_{br} \cdot c_{dr}(p, D) \cdot p/p_b \cdot (t_o + t_p/p)$, also increases with pipeline depth for two reasons. First, the ROB accumulates more instructions when increasing pipeline depth, and by consequence, draining more instructions takes longer. Second, draining instructions with a deeper pipeline also increases drain time (counted in cycles). In other words, the impact of pipeline depth on the branch misprediction penalty is variable and is larger than its effect on the pipeline refill term; in addition, the branch resolution time also increases with deeper pipelines. Figure 18 in the top graph further illustrates this finding: If the pipeline refill term were the only contributing term, then the slopes for the branch misprediction TPI and the I-cache TPI curves would be identical; however, the slope for the branch misprediction TPI curve is much steeper than the slope for the I-cache TPI curve.

Long-latency back-end TPI. The long back-end D-cache TPI term is affected by pipeline depth in two major ways. First, the cache miss latency term may be partially pipelined or nonpipelined. Like the I-cache TPI terms, if (partially) pipelined, this term increases in actual time because of pipelining overhead; if nonpipelined, this term remains constant irrespective of pipeline depth. Second, the number of nonoverlapping long load misses m_{dL2} may decrease with deeper pipelines because of the larger ROB sizes for balanced processor designs. In other words, more Memory-Level Parallelism (MLP) gets exposed. Figure 18 illustrates this very well. The bottom graph assuming nonpipelined cache misses shows that the long-latency load TPI decreases with pipeline depth because of increased MLP. The top graph assuming pipelined cache misses shows that the increased MLP is offset by the increased pipelining overhead: Initially the long-latency TPI decreases and then increases with increased pipeline depth.

Pipelined versus nonpipelined cache misses. Another observation from Figure 18 is that the optimum pipeline depth assuming nonpipelined cache misses is slightly larger than assuming pipelined cache misses. The reason is that the I-cache miss TPI remains constant and the long-latency load TPI decreases with deeper pipelines for nonpipelined cache misses, which makes the optimum pipeline depth shift towards deeper pipelines.

Comparison to prior work on pipeline depth. Prior work on optimal pipeline depths use detailed simulation [Kunkel and Smith 1986; Agarwal et al. 2000; Hrishikesh et al. 2002; Sprangle and Carmean 2002], or use modeling [Emma and Davidson 1987; Dubey and Flynn 1990; Srinivasan et al. 2002; Hartstein and Puzak 2002]. The pipeline depth model closest to our mechanistic model is the model by Hartstein and Puzak [2002]. Hartstein and Puzak divide the total execution time in busy time T_{BZ} and nonbusy time T_{NBZ}. The busy time refers to the time that the processor is doing useful work, that is, instructions are issued; the nonbusy time refers to the time that execution is stalled due to miss events (called hazards by Hartstein and Puzak [2002]). Hartstein and Puzak derive that the total execution time equals busy time plus nonbusy time. We have

$$T = N_{total}/\alpha \cdot (t_o + t_p/p) + N_H \cdot \gamma \cdot (t_o \cdot p + t_p) \tag{10}$$

with N_H the number of miss events; α and γ are empirically derived by fitting the model to data generated with detailed simulation. Comparing Eq. (10) against Eq. (9), we observe that the mechanistic interval model provides more insight than the model of Hartstein and Puzak. For example, according to Hartstein and Puzak [2002], the parameter α is not the superscalar issue width, but the "actual degree of superscalar processing." The interval model shows that α in fact is the achieved dispatch efficiency, which is a function of processor width and the number of intervals. Similarly, Hartstein and Puzak [2002] denote the parameter γ as the fraction of the total pipeline delay stalled due to a miss event averaged over all miss events. In other words, Hartstein and Puzak [2002] assume that all miss event types affect performance the same way by lumping their effects into a single parameter. The mechanistic model, on the other hand, factors out the different types of miss events. One consequence of separating miss event effects in the mechanistic model is that it enables modeling partial pipelining of cache misses, which is impossible using the more empirical Hartstein and Puzak model.

4.2.2 *Pipeline Width.* We now discuss how the various TPI components are affected by width. The main performance advantage of increasing width D of superscalar processing comes from the first term of Eq. (9): Doubling the processor width halves the execution time for performing useful work. This is also illustrated in Figure 19, which shows how the various TPI components are affected by processor width, averaged across all benchmarks. The second term that decreases with increasing width is the long-latency load miss term. The number of nonoverlapping long back-end misses decreases with increasing widths, the reason being the increased ROB size for balanced processor designs.

There are two terms that increase with increasing processor width: the dispatch inefficiency term and the branch misprediction term. The most significant term is the branch misprediction term, which increases because of window drain time. Recall that for a balanced processor design, the ROB accumulates more instructions with an increased processor width and draining more instructions

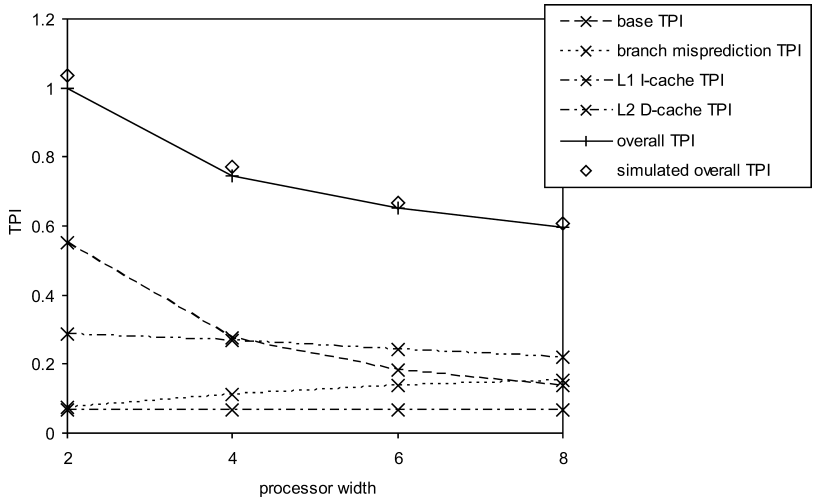

Fig. 19. TPI components and overall TPI as a function of width.

from the ROB takes longer; the increased processor width does not help the drain time because the ROB cannot be drained faster than the instructions' critical path length.

4.2.3 *Branch Mispredictions versus Cache Misses.* The previous discussion on pipeline depth and width performance scaling illustrates that branch mispredictions pose a fundamentally more difficult problem than cache misses. This is apparent from the graphs shown in Figures 18 and 19, which show that the branch misprediction TPI increases with depth and width whereas other TPI components either increase at a slower rate (due to pipelining only), remain constant, or even decrease as a function of width and depth. The root cause for the problematic scaling behavior of branch mispredictions is the branch resolution time. As mentioned before, the branch resolution time increases with depth because of pipelining overhead. In addition, and more importantly, it also increases with width because the ROB contains more instructions when a branch misprediction occurs (because the ROB is bigger in a balanced processor design) resulting in a longer branch resolution time.

This problematic performance scaling relationship of branch mispredictions has an important implication to processor design. It implies that if a designer wants to scale the pipeline depth or width by a given factor, he/she must scale the branch prediction accuracy by more than a linear factor in order to keep the fraction of the time where useful work is done constant. The cache miss rate, on the other hand, must only scale linearly (or even less than linearly for long-latency back-end misses) with depth and width.

4.2.4 *Optimal Pipeline Depth / Width for Balanced Processors.* Having separately discussed the effect of pipeline depth and width on overall performance, we now discuss the relationship between pipeline depth and width. Figure 20

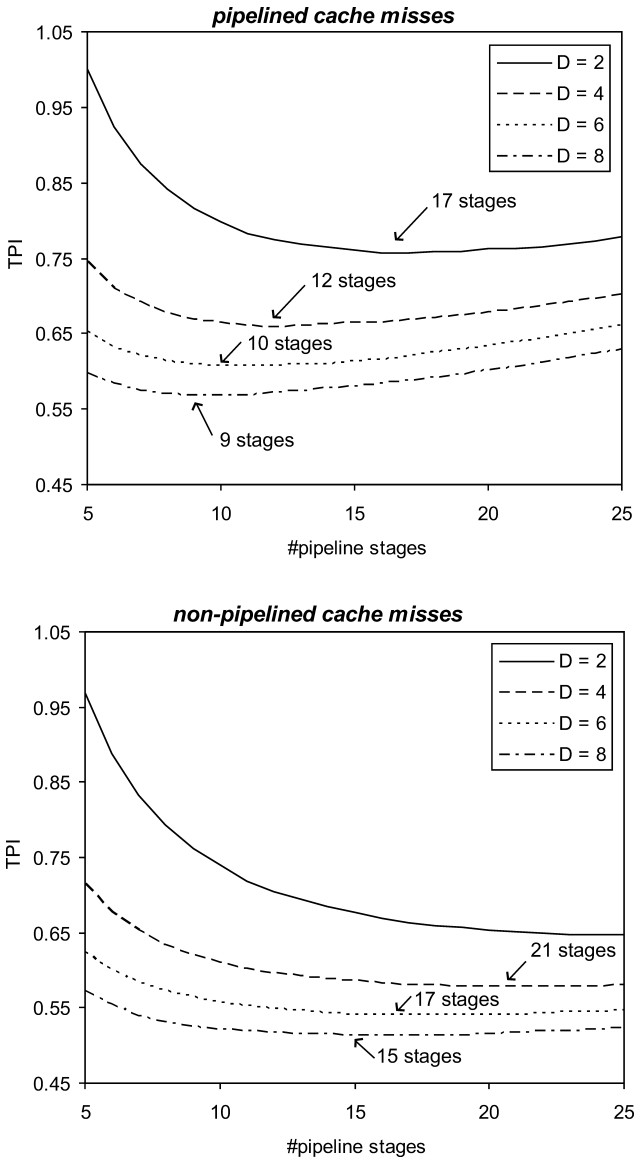

Fig. 20. Overall TPI as a function of depth and width, assuming pipelined cache misses (top graph) and nonpipelined cache misses (bottom graph).

shows overall TPI as a function of pipeline depth for different processor widths. These graphs clearly show that the optimum pipeline depth shifts towards shallower pipelines for increasing processor widths. The reason for the phenomenon is that the base TPI component decreases with increasing width, while the miss event penalty components do not. This inverse relationship between processor width and pipeline depth for optimal performance can be theoretically derived by differentiating Eq. (9) to p. This yields a formula for

the optimal pipeline depth p^*, assuming fully pipelined cache misses

$$p^* = \sqrt{\frac{t_p \cdot t_b}{t_o} \cdot \frac{\frac{N_{total}}{D} + \frac{D-1}{2D} \cdot (m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W))}{m_{iL1} \cdot c_{iL1} + m_{iL2} \cdot c_{L2} + m_{br} \cdot (c_{dr} + c_{fe}) + m_{dL2^*(W)} \cdot c_{L2}}}; \quad (11)$$

or, assuming nonpipelined cache misses

$$p^* = \sqrt{\frac{t_p \cdot t_b}{t_o} \cdot \frac{\frac{N_{total}}{D} + \frac{D-1}{2D} \cdot (m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W))}{m_{br} \cdot (c_{dr} + c_{fe})}}. \quad (12)$$

The preceding formulae can be simplified to the following

$$p^* = \sqrt{\frac{c_1}{D} + c_2}, \quad (13)$$

with the constants c_1 and c_2 being proportional to

$$c_1 \propto N_{total} - \frac{\sum m_i}{2}; c_2 \propto \frac{\sum m_i}{2}. \quad (14)$$

Since the number of miss events $\sum m_i$ is generally much smaller than the total number of dynamically executed instructions N_{total}, it follows that $c_1 \gg c_2$. After eliminating the relatively insignificant c_2 term, the optimal pipeline depth for a given processor width p^* is approximately proportional to the processor width D to the power $-1/2$, namely,

$$p^*(D) \cdot \sqrt{D} \simeq constant. \quad (15)$$

Note that this relationship holds for both pipelined as well as nonpipelined cache miss handling. In other words, when increasing the processor width by a factor c, one must decrease the pipeline depth by a factor $\sqrt{c}$ in order to achieve the optimum depth for the given width. Figure 20 validates this finding. For example, for the pipelined cache miss case, the optimal pipeline depth for the 8-wide machine (9 stages) is approximately a factor $\sqrt{2}$ smaller than the optimal pipeline depth for the 4-wide machine (12 stages); and, the optimal depth for the 4-wide machine is approximately a factor $\sqrt{2}$ smaller than the optimal pipeline depth for the 2-wide machine (17 stages). We obtain a similar finding for the nonpipelined cache miss case.

It is interesting to note that the constant of proportionality between processor width D and the optimum processor depth p^* is different for the pipelined cache miss case (around 24, i.e., optimum pipeline depth of 12 stages for a 4-wide processor) and the nonpipelined cache miss case (around 42, i.e., optimum pipeline depth of 21 stages for a 4-wide processor). Generally speaking, this constant of proportionality is inversely proportional to the pipelined miss event cycles. This means that reducing the number of pipelined miss events increases the optimum pipeline depth (for a given width).

4.3 Overprovisioned Designs

Thus far, we have assumed that out-of-order designs are balanced according to the definition we gave earlier. The definition is based on providing adequate

resource requirements to sustain maximum performance under ideal (no miss event) conditions. This definition has intuitive appeal because sustaining the peak dispatch/issue rate would seem to be the most demanding situation. There are cases, however, when overprovisioning certain resources can yield better overall performance. These situations arise from transient conditions due to the presence of miss events (which are explicitly ignored in our definition of balance).

One situation is the presence of long-latency back-end misses. The model shows that performance can be improved if the number of overlapping long-latency back-end misses is increased. The inhibitor to long-latency back-end miss overlaps, however, is the ROB size. Hence, increasing the ROB size increases the overlap of long misses, which improves performance. This is the principle exploited by proposed processor designs that employ very large ROBs (with appropriate scaling of other associated structures). Runahead processors [Mutlu et al. 2003], kilo-instruction processors [Cristal et al. 2004], and continual flow pipelines [Srinivasan et al. 2004] tolerate long-latency loads by unblocking the ROB being blocked by a long-latency load allowing the processor to execute ahead, and thereby exposing memory-level parallelism. Results in prior papers already indicate the significant performance potential of this approach, and, therefore, we do not consider it further here.

Through the insights gained from the proposed mechanistic model we have discovered another form of overprovisioning that leads to improved performance. Because this method has not been described and evaluated previously, we give it some attention here.

First, recall that the processor width parameter D used in the model refers to dispatch width, and *not* issue width. We have been assuming an effective issue width that is (at least) as large as the dispatch width, but from a dispatch-oriented perspective, one could argue that the issue width this wide is only important for draining the window as fast as it is filled in the absence of miss events (in steady state). For short intervals, however, this window fill/drain balance is not critical.

This observation suggests that, for a given effective issue width, making dispatch wider than the processor's effective issue width might improve performance. This is illustrated in Figure 21. In this example, we set the effective issue width I_{eff} equal to the dispatch width D, and assume one miss event every 96 instructions. Then issue rate is plotted as a function of time. The achieved issue rate ramps up until all 96 instructions are dispatched; then the processor starts draining the window. As one would expect, increasingly wider processors improve performance (fewer cycles on the horizontal axis). However, the peak issue rate is never reached for widths 6 and 8. If issue rate were the determining factor, these two should have the same performance because issue rate is not a constraint. Instead, the performance benefit comes from the wide dispatch. Intuitively speaking, in the case of branch mispredictions, a wider dispatch means that more ILP is exposed in the window sooner, so that a mispredicted branch is resolved sooner. A similar effect happens for instruction cache misses: A wider front-end means that the next I-cache miss is reached sooner.

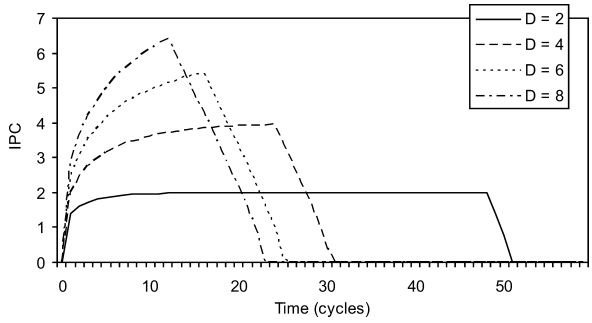

Fig. 21. An example illustrating the benefit of a wide processor: Performance improves while increasing the processor's dispatch width D.

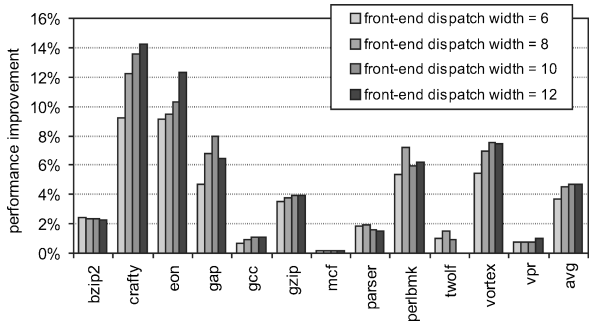

Fig. 22. Performance improvement by making the front-end pipeline wider than issue width: Issue width is fixed to 4-wide while scaling the dispatch width.

In order to validate the merits of a wide dispatch processor, we set up the following experiment. We start with a ROB size, dispatch, and effective issue width that are balanced. We increase the width of the front-end pipeline (as characterized by the dispatch width) while keeping the ROB size and the effective issue width fixed; we assume a fetch engine that is aggressive enough to sustain dispatch close to the maximum dispatch rate. For a baseline 4-wide issue processor, the percentage performance improvement is shown in Figure 22. On average, scaling the front-end pipeline to 6-wide yields an average 3.7% performance improvement; scaling to 8-wide yields an average 4.5% performance improvement. Some benchmarks show significant speedups up to 9% when making the front-end pipeline 6-wide for a 4-wide issue machine.

After detailed analysis, we observe that the speedup comes primarily from reducing the execution time of L1 I-cache miss intervals. Branch misprediction intervals also benefit from a wider front-end, but to a lesser extent: Instructions are dispatched into the ROB more quickly, but with a wider front-end pipeline, the ROB accumulates more instructions which also increases branch resolution time; a net performance improvement is observed, though. The fact that for some benchmarks we observe a slight performance decrease while widening the front-end pipeline comes from the increased branch predictor's delayed update and I-cache prefetching effects along speculated paths.

5. CONCLUSION

In this article we proposed a mechanistic model for estimating out-of-order processor performance. The mechanistic model builds on interval analysis which breaks the total execution time into intervals based on miss events. Overall processor performance can be analyzed and estimated by aggregating the performance of individual intervals with different types and lengths. The major advance of the interval model over prior analytical modeling work is that: (i) it exposes the impact of interval length on both the dispatch efficiency (the more intervals, the poorer the dispatch efficiency) and the branch misprediction resolution time (the smaller the interval, the smaller the branch resolution time), and (ii) it provides insight in the fundamental interactions that take place in the processor. The average difference for the mechanistic interval model compared to simulation is 7% for a 4-wide superscalar out-of-order processor, and we found the model to be accurate across the processor design space.

Using the mechanistic model as a guide, we studied resource scaling in out-of-order processors. In particular, we explored pipeline configuration scaling (depth, width, and their relationship) for balanced processor designs as well as overprovisioned designs. For balanced designs, the mechanistic model provides various insights into how overall performance is affected by the pipeline configuration in terms of the individual contributing TPI components. In particular, we studied how optimum pipeline depth is affected by width and derive the result that for optimal performance, the depth times the square root of the width is nearly constant. As an example of an overprovisioned processor design, we explored the performance potential of wide front-end dispatch processors which improve performance by making the dispatch width wider than the issue width, the reason being that the processor gets to the next miss event sooner. Our experimental results show that for some benchmarks performance can be improved upto 9% by making the front-end pipeline 6-wide instead of 4-wide for a 4-wide issue processor.

As part of our future work, we plan on extending the proposed model along three avenues. First, we plan on extending the modeling of the memory hierarchy by considering memory bandwidth limitations and hardware prefetching effects. Limited memory bandwidth basically results in, and can be modeled as, increased latency for memory requests. Hardware prefetching can be modeled: (i) as a reduced miss rate if timely and useful, (ii) as reduced latency if not timely but useful, or (iii) as an increased miss rate if not useful. Second, we plan on extending the model for nonbalanced processor designs. One example of a nonbalanced design, as discussed before, is the large-window processor, such as a runahead processor [Mutlu et al. 2003], a kilo-instruction processor [Cristal et al. 2004], and a continual flow pipeline [Srinivasan et al. 2004], which strive at exploiting distant MLP. Extending the proposed model to large-window processors would require that we model overlap effects between front-end and back-end miss events, as they may have a more significant impact on overall performance than is the case for balanced processor designs. Third, we plan on extending the model towards multiprocessors in general, and

chip-multiprocessors in particular. This will involve the modeling of coherence traffic and resource conflicts in shared resources.

REFERENCES

AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 248–259.

BERG, E. AND HAGERSTEN, E. 2005. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, 169–180.

BROOKS, D., MARTONOSI, M., AND BOSE, P. 2000. Abstraction via separable components: An empirical study of absolute and relative accuracy in processor performance modeling. Tech. rep. RC 21909, IBM Research Division, T. J. Watson Research Center. December.

BURGER, D. C. AND AUSTIN, T. M. 1997. The SimpleScalar tool set. *Comput. Architecture News*. See also `http://www.simplescalar.com` for more information.

CHOU, Y., FAHS, B., AND ABRAHAM, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 76–87.

CRISTAL, A., SANTANA, O. J., VALERO, M., AND MARTINEZ, J. F. 2004. Toward kilo-instruction processors. *ACM Trans. Architecture Code Optimiz. 1*, 4, 389–417.

DUBEY, P. K., ADAMS III, G. B., AND FLYNN, M. J. 1994. Instruction window size trade-offs and characterization of program parallelism. *IEEE Trans. Comput. 43*, 4, 431–442.

DUBEY, P. K. AND FLYNN, M. J. 1990. Optimal pipelining. *J. Parallel Distrib. Comput. 8*, 1, 10–19.

EECKHOUT, L. AND DE BOSSCHERE, K. 2001. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 25–34.

EMMA, P. G. 1997. Understanding some simple processor-performance limits. *IBM J. Res. Development 41*, 3, 215–232.

EMMA, P. G. AND DAVIDSON, E. S. 1987. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Trans. Comput. 36*, 7, 859–875.

EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. 2006a. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 175–184.

EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. 2007. A top-down approach to architecting CPI component performance counters. *IEEE Micro 17*, 1, 84–93.

EYERMAN, S., SMITH, J. E., AND EECKHOUT, L. 2006b. Characterizing the branch misprediction penalty. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 48–58.

FIELDS, B. A., BODIK, R., HILL, M. D., AND NEWBURN, C. J. 2004. Interaction cost and shotgun profiling. *ACM Trans. Architecture Code Optimiz. 1*, 3, 272–304.

GLEW, A. 1998. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session*.

GUO, F. AND SOLIHIN, Y. 2006. An analytical model for cache replacement policy performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, 228–239.

HARTSTEIN, A. AND PUZAK, T. R. 2002. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 7–13.

HARTSTEIN, A. AND PUZAK, T. R. 2003. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 117–126.

HRISHIKESH, M. S., JOUPPI, N. P., FARKAS, K. I., BURGER, D., KECKLER, S. W., AND SHIVAKUMAR, P. 2002. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 14–24.

IPEK, E., MCKEE, S. A., DE SUPINSKI, B. R., SCHULZ, M., AND CARUANA, R. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 195–206.

JOSEPH, P. J., VASWANI, K., AND THAZHUTHAVEETIL, M. J. 2006a. Construction and use of linear regression models for processor performance analysis. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 99–108.

JOSEPH, P. J., VASWANI, K., AND THAZHUTHAVEETIL, M. J. 2006b. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 161–170.

KARKHANIS, T. AND SMITH, J. E. 2002. A day in the life of a data cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI) held in conjunction with ISCA*.

KARKHANIS, T. AND SMITH, J. E. 2007. Automated design of application specific superscalar processors: An analytical approach. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 402–411.

KARKHANIS, T. S. AND SMITH, J. E. 2004. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 338–349.

KUNKEL, S. AND SMITH, J. E. 1986. Optimal pipelining in supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, 404–411.

LEE, B. AND BROOKS, D. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 185–194.

MICHAUD, P., SEZNEC, A., AND JOURDAN, S. 1999. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2–10.

MICHAUD, P., SEZNEC, A., AND JOURDAN, S. 2001. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Internal J. Parallel Program. 29*, 1.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 129–140.

NOONBURG, D. B. AND SHEN, J. P. 1997. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, 52–62.

NOONBURG, D. B. AND SHEN, J. P. 1994. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA)*, 298–309.

RISEMAN, E. M. AND FOSTER, C. C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Trans. Comput. C-21*, 12, 1405–1411.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 45–57.

SORIN, D. J., PAI, V. S., ADVE, S. V., VERNON, M. K., AND WOOD, D. A. 1998. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, 380–391.

SPRANGLE, E. AND CARMEAN, D. 2002. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 25–34.

SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 107–119.

SRINIVASAN, V., BROOKS, D., GSCHWIND, M., BOSE, P., ZYUBAN, V., STRENSKI, R. N., AND EMMA, P. G. 2002. Optimizing pipelines for power and performance. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, 333–344.

TAHA, T. M. AND WILLS, D. S. 2003. An instruction throughput model of superscalar processors. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP)*, 156–163.

TAHA, T. M. AND WILLS, D. S. 2008. An instruction throughput model of superscalar processors. *IEEE Trans. Comput. 57*, 3, 389–403.

WALL, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 176–188.

ZHONG, Y., DROPSHO, S. G., AND DING, C. 2003. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Received February 2008; accepted February 2009