Appears in the 31st International Symposium on Computer Architecture

A First-Order Superscalar Processor Model

Tejas S. Karkhanis James E. Smith
Univ. of Wisconsin – Madison, Dept. of Electrical and Computer Engineering
{karkhani,jes}@ece.wisc.edu

Abstract

A proposed performance model for superscalar processors consists of 1) a component that models the relationship between instructions issued per cycle and the size of the instruction window under ideal conditions, and 2) methods for calculating transient performance penalties due to branch mispredictions, instruction cache misses, and data cache misses. Using trace-derived data dependence information, data and instruction cache miss rates, and branch miss-prediction rates as inputs, the model can arrive at performance estimates for a typical superscalar processor that are within 5.8% of detailed simulation on average and within 13% in the worst case. The model also provides insights into the workings of superscalar processors and long-term microarchitecture trends such as pipeline depths and issue widths.

1. Introduction

Superscalar processor performance is very often evaluated via detailed simulation. Although accurate, this method is also time-consuming, both when creating the simulation model and when running the simulations. Furthermore, although a simulation model can generate a lot of data, it often falls short in providing insight regarding what is going on inside the processor.

An alternative to simulation is analytical modeling. Analytical models have clear speed advantages, but also, if well-constructed, they can provide valuable insight. To date, however, superscalar processor models have been used rather infrequently, presumably because the complexity of superscalar processors has limited the accuracy of analytical models. In this paper, we propose and evaluate an approach to superscalar processor modeling that is intuitive, provides insight, and is reasonably accurate. The proposed model consists of an analytical core that incorporates cache and branch predictor statistics gathered from functional-level trace driven simulation.

1.1 Model Approach

In the work presented here, we develop a first-order model that provides good accuracy and is capable of providing the insight we desire. Then, future continuation research can target additional features and refinements to fill out a complete and more accurate superscalar model. The first-order superscalar processor that we model has a single, homogenous instruction issue window. Instructions issue out-of-order in oldest-first priority. The reorder buffer is a separate structure (not combined with the issue window as in an RUU[1]). The pipeline width, issue width, and retire width are the same and are parameterized. The front-end pipeline depth can also be adjusted. Caches and branch predictors are included in the model, but features like prefetching are not. There is an unbounded number of functional units of each type.

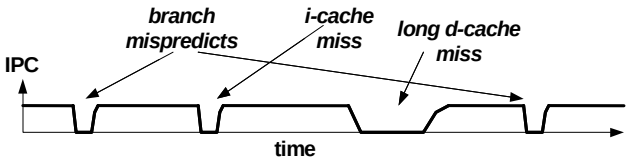

Figure 1. Useful instructions issued per cycle (IPC) as a function of time.

The basis for the model development to follow is illustrated in Figure 1. The figure shows a graph of performance, measured in useful instructions issued per cycle (IPC), as a function of time. A superscalar processor sustains a constant background performance level, punctuated by transients where performance falls below the background level. These transient events are caused by branch mispredictions, instruction cache misses, and data cache misses – referred to collectively as *miss-events*. Overall performance is calculated by first determining the sustained performance under ideal conditions (i.e. with no miss-events) and then subtracting out performance losses caused by the miss-events.

To provide initial support for this basic approach, we simulated a baseline processor that has five front-end pipeline stages, an issue width of four, a window size of 48 entries, and a reorder buffer with 128 entries. The instruction and data caches are 4K 4-way set associative with 128 bytes per cache line; a unified 512K L2 cache is 4-way set-associative with 128 byte lines, and the branch predictor is 8K gShare. We ran the following five sets of simulations: 1) everything ideal: i.e. ideal caches and ideal branch predictor, 2) "real" caches and branch predictor, 3) everything ideal except for the branch predictor, 4) everything ideal except for instruction cache, 5) everything ideal except for data cache. We next evaluated net performance losses for each of the three types of miss-

events *in isolation*. That is, we computed total clock cycles for simulation 3 minus total clock cycles for simulation 1 to arrive at the time penalty due to branch mispredictions. Similarly, we determined the time penalties for the cache misses using simulations 1, 4 and 5.

The three independently-derived performance penalties were then added to the ideal time. The resulting performance is compared with the fully "realistic" simulation 2. As a second comparison, during simulation 2 we counted the fractions of branch mispredictions and instruction cache misses that overlap a data cache miss and compensated for them by ignoring the penalty for branches and i-cache misses that overlap a d-cache miss.

The performance results, converted to IPC, are given in Figure 2. For each of the SPECint benchmarks, the three bars are 1) *combined*: the "realistic" performance, 2) *independent*: the performance determined by adding each of the independently-determined miss-event penalties to the ideal performance, and 3) *overlaps compensated:* the same as 2), but with compensation for overlaps with data cache misses. The accuracy of the estimation method is quite good, across the board. The fully independent approximation is quite good (middle bar). The average error is 5%, and the highest errors are 16% (*twolf*) and 10% (*gzip*). Overlap compensation improves accuracy only slightly; the average error is 4%, and the highest error is 10% (*gzip*).

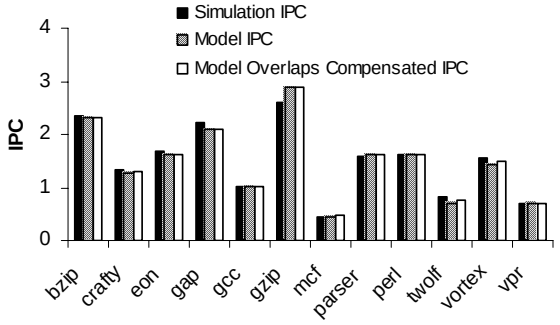

Figure 2: Demonstration of relative independence of miss-events with respect to performance.

The independence of miss-event penalties provides a powerful lever for constructing a superscalar model because it allows us to reason about, and model, each category of miss-event more-or-less in isolation. Note, however, that the individual miss-events within the same category are not necessarily independent, implying that we may have to account for "bursts" of miss-events of a given type, e.g. when a burst of branch mispredictions or cache misses cluster together closely in time.

In the remainder of the paper we develop a model that contains the following components:

1) A method for determining the ideal, sustainable performance (IPC) in terms of implementation-independent dynamic instruction stream statistics and microarchitecture parameters.

2) Methods for estimating the penalties for branch predictions, instruction cache misses, and data cache misses, in terms of the microarchitecture parameters.

3) A method for taking miss-event rates and combining them with the information from 1) and 2) to arrive at overall performance estimates.

Along the way, we use the model to derive insights into the operation of superscalar processors. Finally, given the complete model, we demonstrate its usefulness for evaluating important microarchitecture trends.

1.2 Related Work

Emma and Davidson [2], present an early theoretical method for analyzing in-order pipelines. They characterize the effects of branch and data dependences on processor performance. More recently, Hartstein and Puzak [3] and Sprangle and Carmean [4], present analytical models for determining the optimal front-end pipeline depth for both in-order and out-of-order superscalar processors. In [3] two important parameters of the analytical model are measured via detailed simulation: degree of superscalar processing and the fraction of stall cycles per pipeline stage. In [4] the analytical model accounts for relative performance loss due to microarchitecture "loops", such as a branch misprediction loop. A detailed simulation is performed for the baseline case, and then the effect of increasing the processor front-end is simulated. From these two simulations the performance degradation for every cycle increase in the branch misprediction loop is computed and becomes a parameter in the model. In both models [3] [4] detailed superscalar simulations are required for model parameters. Our model avoids detailed superscalar simulations and relies on instruction trace analysis for model parameters.

Noonburg and Shen [5] develop a concise model based on probability matrices. Their approach is to first limit the parallelism because of control flow, then because of inefficient fetch, and then because of data dependencies. Our approach is to first consider ideal IPC using pure data dependences and adjust for performance losses due to the miss-events. A limitation of the model [5] is that it does not include the effect of the re-order buffer and data cache misses.

Michaud et al.[6, 7], develop an analytical model for expressing Instruction Level Parallelism (ILP) as a function of the window size in superscalar processors. Their goal is to gain insight in the relationship between issue width and fetch width. Their expression for ILP versus issue window size is similar to one part of our analytical model. Our model covers the whole processor, not just the front-end; and it includes models for branch misprediction and data cache miss penalties.

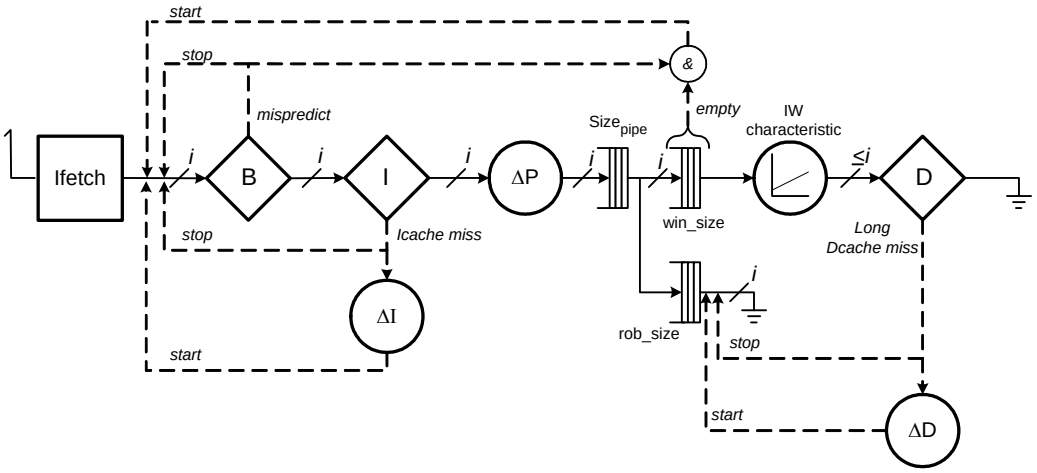

Figure 3. Schematic drawing of proposed superscalar model. Solid lines indicate instruction flow; dashed lines indicate "throttles" of instruction flow due to miss-events.

Statistical simulation methods [8-11] collect many of the same program statistics as used by our model, and use them to generate a synthetic trace that drives a simple superscalar simulator. In effect, our model performs statistical simulation, *without the simulation*, and overall accuracy is similar.

Sorin et al. [12] present a model for shared memory multiprocessors based on mean-value analysis. They "black-box" the processor, L1, and L2 caches and model the memory traffic beyond the L2 cache. Our model includes the superscalar processor core. Fields et al. [13] also assume a blackbox viewpoint of the processor and evaluate correlations observed among simulation results. They assign a cost to interactions among performance degrading events, and then use the cost to find the primary bottleneck. We skip the simulation step and instead analyze instruction traces to get statistics for the superscalar processor core model.

Ofelt [14] proposes a profile-based performance prediction technique for out-of-order superscalar processors that goes through a lightweight instrumentation phase followed by analysis phase. Their technique accounts for instruction cache and branch misprediction effects, but not data cache effects. Our model accounts for instruction cache, branch misprediction, and data cache effects.

2. Top-level Model

For reasoning about superscalar processor operation, we use a schematic representation as shown in Figure 3. The Ifetch unit is capable of providing a never-ending supply of instructions. Instructions pass through the front-end pipeline, experiencing a ΔP delay, before being dispatched into both the issue window and the re-order buffer. The fetch width, pipeline width, dispatch width, retire width, and maximum issue width are all characterized with parameter *i*. Instructions issue from the window at a rate determined by the *IW characteristic*, i.e. a function that determines the number of instructions that ***I***ssue in a clock cycle, given the number of instructions in the ***W***indow.

At the time instructions are fetched, there is a probability, B, that there is a branch misprediction. If this happens, the fetching of useful instructions is stopped. Fetching of useful instructions resumes only when the issue window becomes empty of useful instructions. Also at the time instructions are fetched, there is a probability, I, that there is an instruction cache miss. If there is a miss, instruction fetching is stopped and resumes only after the instructions can be fetched from the L2 cache, or memory; this is modeled by delay ΔI. When there is a long data cache miss (L2 miss) the retirement of instructions from the reorder buffer is stopped. After a miss delay, ΔD, data returns from memory, and retirement is restarted. Short data cache misses (L1 misses) are modeled as if they are handled by long latency functional units.

The model implies that the penalties from branch mispredictions and instruction cache misses will serialize. However, long data cache misses may overlap with branch mispredictions, with instruction cache misses, and with each other.

The expression for overall performance is given in equation(1), where *CPIsteadystate* is the background sustainable performance when there are no miss-events. *CPIbrmisp, CPIicachemiss,* and *CPIdcachemiss* are the additional CPI due to branch misprediction events, instruction cache miss-events and data cache miss-events, respectively.

$$CPI = CPI_{steadystate} + CPI_{brmisp} + CPI_{icachemiss} + CPI_{dcachemiss} \quad (1)$$

Note that here we give performance in CPI – in other places we convert to IPC – throughout our discussion, we use either of the two, depending on which is more appropriate at the time it is being used.

3. IW Characteristic

The IW characteristic expresses the relationship between the number of instructions in the issue window and the number of instructions that will issue (on average). The IW characteristic is important both for determining the ideal, sustained performance level and for estimating the penalties of miss-events.

In one of the very first studies on instruction level parallelism [15] Riseman and Foster observed (in today's terms) that the number of instructions that can issue per cycle is roughly the square root of the number of instructions in the window. More recently, Michaud, Seznec and Jourdan [7] similarly observe that the IW characteristic follows a Power-Law relationship and provide an insightful analysis of the phenomenon and its relationship to instruction delivery. They show that the slope of the Power-Law line on a log-log scale is approximately 0.5 for their benchmarks, indicating a square-root relationship.

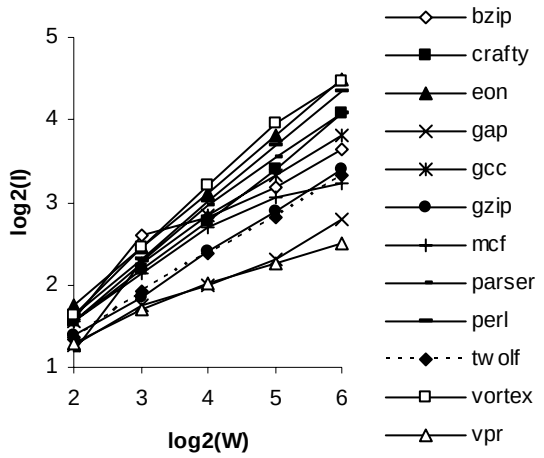

Figure 4: Power-Law relationship between the issue window size and the issue width.

Starting with dependence statistics taken from instruction traces (as in [8, 10]]), the points on the IW curve for the unlimited issue case can be characterized by a set of relatively complex simultaneous, non-linear equations. A detailed discussion of the analytical derivation of the IW characteristic is beyond our scope. A practical alternative of similar complexity is to perform idealized (no miss-events) trace-driven simulations with an unlimited number of *unit-latency* functional units and unbounded issue width. The only thing that is limited is the issue window size. IW curves generated in this way are given in Figure 4.

These initial IW curves are essentially implementation independent; they depend only on the basic register-based data dependence properties of the benchmark. For a specific implementation with limited issue width and non-unit latencies, we generate the IW characteristics in the following way. First, because they have a Power-Law relationship, we fit the IW curves to the line $I = \alpha W^{\beta}$ The values of α and β for three illustrative benchmarks are given in Table 1. These benchmarks are at the two extremes (*vortex* and *vpr*) and in the middle (*gzip*) of the curves shown in Figure 4. Figure 5 compares the IW curves for these benchmarks as given in Figure 4 with the computed linear fit.

Table 1: Power-Law parameters for unit-latency case.

Bmk.	α	β	Avg. Lat.
gzip	1.3	0.5	1.5
vortex	1.2	0.7	1.6
vpr	1.7	0.3	2.2

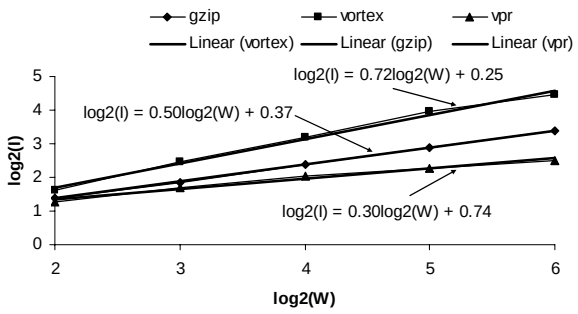

Figure 5: Linear IW curve fit for illustrative benchmarks.

To account for non-unit latencies we apply Little's Law. If the average issue rate is I_1 with a window size of W and unit functional unit latencies, then the average time spent in the window by a given instruction is $T=W/I_1$. Or, $I_1=W/T$. If the average instruction latency is L, then all dependence chains, weighted by latencies, are approximately L times longer than for the unit latency case. This means that the time spent in the window for the average instruction will be L times longer than for unit latency, so the issue rate with average latency L, can be easily derived as: $I_L = I_1/L$. That is, for a given number of instructions in the window, the average issue rate is the unit latency issue rate divided by the average latency. The last column of Table 1 gives the average instruction latencies for the three illustrative benchmarks.

When the maximum issue width is limited, as it would be in a superscalar processor, then the IW curves change somewhat [16]. For example, by using simulation with limited issue width, we arrive at the IW characteristic in Figure 6. The limited issue curves follow the ideal curves until the window size equals the maximum issue width, and then they asymptotically approach the issue width limit; that is, instruction issue *saturates* at the maximum rate.

We approximate this behavior by assuming unlimited issue width behavior (following the non-unit latency

power-law curve as just derived) until the issue rate reaches the maximum issue limit. Then, as in Jouppi [16], we assume issue rate saturates at the maximum issue width. As we will see, for our first-order superscalar model, this approximation is adequate. For most benchmarks, we use a window size that is large enough so that the issue rate in absence of miss-events is in the saturation part of the curve.

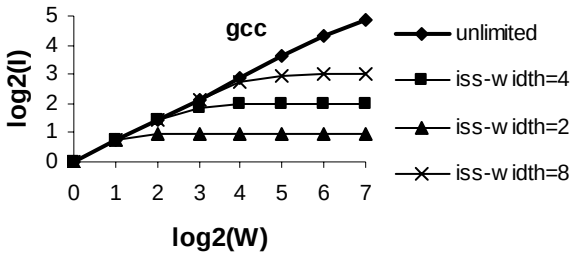

Figure 6: IW characteristic after limiting the issue width.

4. Modeling of Miss-Events

The performance penalties for miss-events are modeled by first determining the penalty for each type of miss-event, counting the numbers miss-events of each type, then multiplying. The miss-event counts are generated via simple trace-driven simulations. The methods for calculating penalties are described in following subsections.

4.1 Branch Misprediction Penalty

To model the penalties for the miss-events, we rely on the schematic in Figure 3 and the IW characteristic. First consider a single branch misprediction in isolation. The transient in the IPC plot (recall Figure 1) is shown in Figure 7. Initially, the processor is issuing instructions at the steady-state IPC. Then a mispredicted branch causes fetching of useful instructions to stop. Eventually, the mispredicted branch enters the instruction issue window. At this point, no more useful instructions enter the window until the mispredicted branch is resolved. If the window issues instructions in oldest-first priority, none of the miss-speculated instructions will inhibit any of the useful instructions from issuing. Consequently, only useful instructions need to be considered.

The IW characteristic allows the determination of the number of issued instructions each cycle as the window is emptied of useful instructions. The first cycle the steady state number of instructions, *i*, will issue. Then, the window will have *W*-*i* instructions (*W* is the number of instructions in the issue window), so fewer will issue on the following cycle, etc. The IPC as the window drains is approximately a straight line (as derived in [6, 7]), and illustrated in Figure 7. Eventually, the mispredicted branch is resolved – we assume that the mispredicted branch is the oldest instruction in the window at the time it is resolved. To validate this assumption, we used detailed simulations which showed that there are only 1.3 useful instructions left in the window when a mispredicted branch issues (averaged over all benchmarks); *gap* is the only outlier with 8 useful instructions still left in the window on average.

After branch resolution, the pipeline is flushed and fetching begins from the correct path. The correct path instructions take front-end pipeline depth cycles, ΔP, to reach the window. Then the window begins filling and instruction issue ramps up, again following points on the IW curve, until it eventually reaches the steady state IPC level. The ramp-up curve rises quickly at first, then more slowly as instructions are issued while the window is filling (like filling a "leaky bucket" [7]).

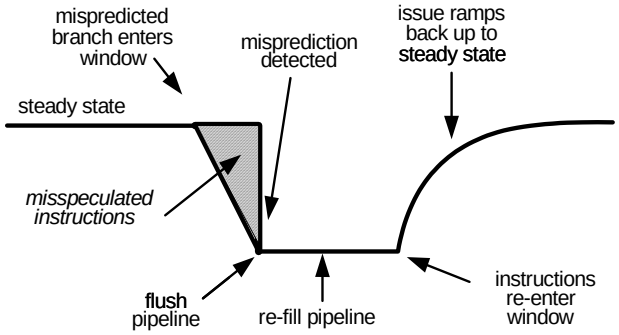

Figure 7: Branch misprediction transient.

Equation(2) expresses the penalty for an isolated branch misprediction, *isolated_brmisp_penalty*; *win_drain* is the penalty for draining the window, Δ*P* is the depth of the front-end pipeline and *ramp_up* is the penalty for ramping up to the steady-state IPC.

$$isolated\_brmisp\_penalty = win\_drain + \Delta P + ramp\_up \quad (2)$$

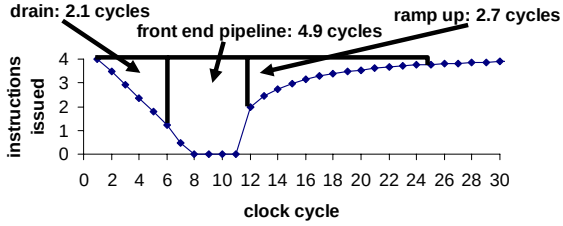

Figure 8: Transient curve for an isolated branch misprediction α=1, β =0.5.

Using Excel, we generated a curve for the branch misprediction transient for a square-law IW characteristic [7](α=1, β =0.5), the average for SpecINT2000 benchmarks once non-unit latencies are accounted for, and a front-end pipeline of five stages (see Figure 8). From this curve, we can estimate the performance losses during

drain, pipeline fill, and ramp-up. If we assume that the branch issues at time 6, at which point there are about 1.4 instructions in the window, then the aggregate drain penalty is 2.1 cycles, found by subtracting the cycles to issue the instructions at the steady state rate from the cycles required for issuing the same instructions while draining. Similarly, the ramp up penalty is computed as 2.7 cycles, and the pipeline fill delay is 4.9 cycles, leading to a total penalty of 9.7 cycles.

This method sets an upper bound penalty for each branch misprediction because it assumes a misprediction occurs *in isolation*. For bursts of branch mispredictions, the drain and ramp-up penalties "bracket" a series of pipeline fills, each of which delivers a small number of useful instructions. In the extreme case of *n* consecutive branch mispredictions, the penalty per misprediction, *brmisp_penalty*, is given by equation(3).

$$brmisp\_penalty = \Delta P + \frac{win\_drain + ramp\_up}{n} \qquad (3)$$

Hence, depending on the amount of clustering of branch mispredictions, for the baseline processor we would expect the penalty to be between 5 and 10 cycles. We observe that *the branch misprediction penalty can be significantly greater than the (often assumed) front-end pipeline depth.* For the example five-stage front end, the total penalty can be twice the front-end pipeline depth.

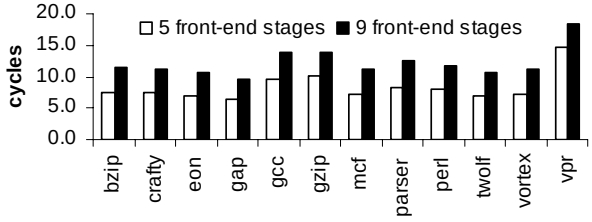

Figure 9: Penalty per branch misprediction for front-end pipelines with 5 and 9 stages.

To evaluate this part of the model, we simulated the baseline processor with both five and nine front-end stages, using ideal instruction and data caches with a realistic branch predictor (8KB gShare). In a second set of simulations, ideal branch predictors are used. Then using the results of these two sets of simulations, the average penalty per branch misprediction is computed. The results are in Figure 9. The y-axis is the penalty in terms of cycles per branch misprediction. For the five stage front-end pipeline, the penalty is typically between 6.4 cycles and 10 cycles, (14.7 cycles for *vpr*). These numbers are greater than the front-end pipeline depth and within the range predicted by the model. Benchmark *vpr* is an outlier because it is the only benchmark with inherently low ideal ILP (indicated by a β of 0.3 in Table 1) and a relatively high average functional unit latency (2.2 cycles). This combination significantly shifts *vpr* away from the assumed (α=1, β=0.5) square-law curve, making its *win_drain* and *ramp_up* longer than those of rest of the benchmarks. Similar to the five stage front-end, with a nine stage front-end pipeline, the penalty for a branch misprediction is greater than nine – as much as 13.8 cycles for *gcc* and *gzip*, (18.3 cycles for *vpr*).

4.2 Instruction Cache Misses

The instruction cache miss transient is illustrated in Figure 10. It has the same basic shape as the branch misprediction transient given above, but some of the underlying phenomena are different. Initially, the processor issues instructions at the steady-state IPC. At the point a miss occurs, there are instructions in the instruction issue window as well as the front-end pipeline. The instructions buffered in the front-end pipeline keep the window filled for a while, but eventually the window drains and the issue-rate drops to zero (following the same curve as for branch mispredictions). After a miss delay, ΔI, instructions are delivered from the L2 cache (or main memory) and begin entering the front-end pipeline. After passing through the pipeline, they eventually reach the instruction issue window. Then, the instruction issue rate ramps up following the IW characteristic.

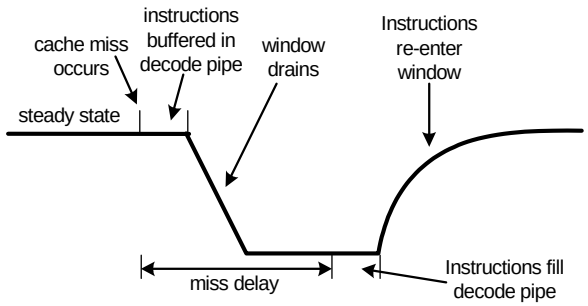

Figure 10: Instruction Cache Miss transient.

Equation(4) gives the penalty, in terms of cycles, for an isolated instruction cache miss.

$$isolated\_icache\_misspenalty = \Delta I + ramp\_up - win\_drain \qquad (4)$$

Consequently, we can make the initial observation: *the instruction cache miss penalty is independent of the front-end pipeline length.* The front-end pipeline can be made arbitrarily deep without affecting the instruction cache miss penalty. The equation also indicates that the *ramp_up* and *win_drain* offset each other (in contrast to the case for branch mispredictions where they add). Referring back to Figure 8 we note that the drain penalty and ramp-up penalty are about the same so their effects cancel. Consequently, we can make a second observation: *the total instruction cache miss penalty is approximately equal to the L2 cache (or main memory) latency*. If there are *n* consecutive instruction cache misses in a burst, then

the equation for the penalty per miss, *icache_misspenalty*, is slightly modified and is given in equation(5).

$$icache\_misspenalty = \Delta I + \frac{ramp\_up - win\_drain}{n} \quad (5)$$

Because *win_drain* and *ramp_up* offset each other (and this number is further diminished when divided by *n*) equation(5) leads to the observation that *an instruction cache miss yields the same penalty regardless of whether it is isolated or is part of a burst of misses*.

To confirm the above observations, we simulated the baseline processor as before, with five and nine front-end pipeline stages. The branch predictors and data caches are ideal, but a non-ideal 4K 4-way set associative instruction cache with 128 byte cache lines is modeled. The instruction cache miss delay (L2 access delay) is set at 8 cycles for both the five and nine stage front-end processors. The same processors with ideal instruction caches are also simulated, and the average penalty per instruction cache miss is computed. The observations derived from the analytical model are supported by the simulation results of Figure 11. In the figure, the y-axis is the penalty (in cycles) for every instruction cache miss. We see that the penalty is approximately 8 cycles (equal to the L2 miss delay) and is independent of the front-end pipeline depth.

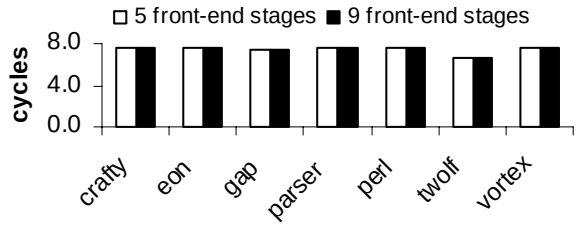

Figure 11: Simulated results show that the icache miss penalty is independent of the front-end pipeline depth. Benchmarks not shown had a negligible number of misses.

4.3 Data Cache Misses

Data cache misses are more complex than instruction cache misses and branch mispredictions, primarily because they can overlap both with themselves and with the other miss-events. Before proceeding further we establish some terminology: *rob_size* is defined as the number of ROB slots, *win_size* is the number of issue window slots, and *dispatch_width* is the maximum number of instructions that can be dispatched into the issue window (and ROB). We begin by dividing data cache misses into two categories: *short misses* – the ones that have latency significantly less than the maximum ROB fill time, i.e. *rob_size/dispatch_width*, and *long misses* -- those whose penalty is significantly greater than the maximum ROB fill time. For the first-order superscalar processor model, L1 cache misses that hit in the L2 cache are short misses, and those that miss in the L2 cache are long misses. Short misses are modeled as if they are serviced by long latency functional units. Therefore, short misses are modeled by their effect on the IW characteristic (and is reflected in the third column of Table 1). This leaves long misses for additional modeling.

If we analyze isolated long data cache misses, there are two events that can potentially trigger performance losses: 1) the window fills with instructions that are dependent (directly or indirectly) on the load that misses, causing instruction issue to stop or 2) the ROB fills because the miss load instruction cannot retire, dispatch stalls, and eventually issue stops.

To determine the relative importance of these two events, we performed the following experiment. We simulated the baseline 4-wide processor with everything ideal except for a 128 KB data cache. The penalty for a data cache miss is set to 200 cycles. In order to study data cache misses in isolation, whenever one miss is already in progress, other data cache misses (if any) are changed to hits. After 200 cycles, instruction issue will certainly be stalled, for whatever the cause, and the instructions remaining in the window at that time are those that are dependent (directly or indirectly) on the load that missed. The simulations showed that the ROB fills and blocks dispatch in virtually every case. After 200 cycles, the window is less than half full (except for *vpr* where 34 window slots are occupied on average). This indicates that blockage due to a full window does not occur to any significant degree when there is a long data cache miss.

Because the ROB filling and causing stall of dispatch is the dominant cause of performance loss when there is a long data cache miss, we develop our model accordingly.

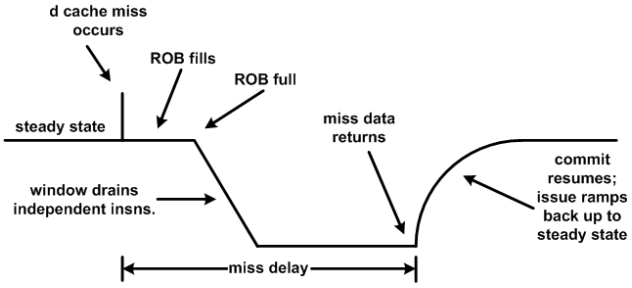

Figure 12: Transient of an isolated data cache miss.

Consider the transient for an isolated long data cache miss given in Figure 12. Initially, the processor is issuing at the steady-state IPC, and a long data cache miss occurs. The issuing of independent instructions continues, and the re-order buffer eventually fills. At that point, dispatch will stall, and when all instructions independent of the load have issued, issue will stall. After miss delay cycles, ΔD, from the time the load miss is detected, the data returns from memory, the missed load commits, and the independent instructions that have finished execution also

commit in program-order. As they commit, room in the ROB opens up, dispatch resumes, and instruction issue ramps up following the IW characteristic.

The expression in equation(6) is the data cache miss penalty, in terms of cycles, for an isolated data cache miss as just described. The parameter *rob_fill* is the number of cycles it takes to fill the re-order buffer after the missed load is issued.

$$isolated\_dcache\_misspenalty = \Delta D - rob\_fill - win\_drain + ramp\_up \quad (6)$$

Because the *ramp_up* and *win_drain* offset each other, the penalty is approximately $\Delta D - rob\_fill$. If the load instruction is the oldest (or nearly so) at the time it issues, then the ROB will already be full (or nearly so), so *rob_fill* is approximately zero, and the penalty will be approximately ΔD. At the other extreme, if the load that misses happens to be the newest instruction in the window, then it will take approximately *rob_size/dispatch_width* cycles to fill the ROB in behind the load, so the penalty will be approximately ΔD - *(rob_size / dispatch_width)*.

The data cache simulation experiment showed that, on average, when a load misses there are 9 instructions ahead of it in the ROB. The outliers are *gap* with 27 instructions ahead of the missed load, and *twolf* and *vpr* with 19 instructions each. Hence, the load that misses is relatively old at the time it issues (at least as a first-order approximation), so we model the data cache miss penalty as ΔD.

The above analysis is for an isolated long data cache miss. To handle overlapped long data cache misses, further analysis is necessary. The overlap case occurs when another data cache miss happens within *rob_size* number of instructions of the first load miss. If this is the case, and the loads are independent (as is most often the case), then their miss penalties will overlap. Figure 13 illustrates the phenomena for two such load misses. Initially the processor is issuing at the sustained IPC. The first load, *ld1*, misses in the data cache. After the load miss, instruction issuing continues until the ROB fills and then issue stops. In the case we are considering, the second load that misses, *ld2*, is one of the instructions that issues before issue stops. Then miss delay, ΔD cycles, after the first load misses, its data returns. Instruction *ld1* and instructions between the *ld1* and *ld2* retire. As they do so, room opens up in the ROB, and a number of instructions equivalent to the number of instructions between the *ld1* and *ld2* are dispatched in the window and the re-order buffer. These instructions issue and then wait in the re-order buffer until the data for the second load miss, *ld2*, returns. Then, *ld2* retires, as do other instructions in the ROB, and issue ramps back up. Assuming the second load miss issues *y* cycles after the first one, equation(7) is the expression for penalty per load miss.

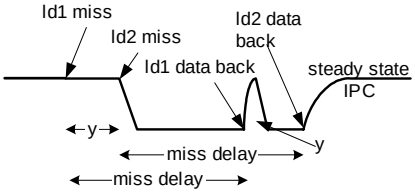

Figure 13: Two loads that experience a long miss are independent of each other and in ROB distance of each other.

$$dcache\_misspenalty = \frac{\left(\begin{array}{l} y + \Delta D - rob\_fill \\ -win\_drain - y + ramp\_up \end{array}\right)}{2} = \frac{isolated\_dcache\_misspenalty}{2} \quad (7)$$

Observe that in the expression in equation(7) the *y* values all cancel, so the combined penalty is half the penalty for an isolated miss and is *independent* of the distance between the two loads that miss; *the only thing that matters is whether they occur within a rob_size number of instructions.*

In general, if N_{LDM} is the number of long data cache misses and $f_{LDM}(i)$ is the probability that misses will occur in groups of i then equation(8) gives the penalty for a long data cache miss, on average.

$$dcache\_misspenalty = isolated\_dcache\_misspenalty \times \sum_{i=1}^{N_{LDM}} \left(\frac{f_{LDM}(i)}{i} \right) \quad (8)$$

The distribution $f_{LDM}(i)$ is collected as a by-product of the instruction trace analysis. We measure the distances between long data cache misses. Then, given a specific *rob_size* and N_{LDM} , the distribution $f_{LDM}(i)$ can be determined. Figure 14 has the penalty for every long data cache miss measured from detailed simulation and the one we computed using the method just described. The model is reasonably close, although not as close as other parts of the model. The handling of data cache misses is one of the more difficult parts of the model and relies on a number of simplifying assumptions.

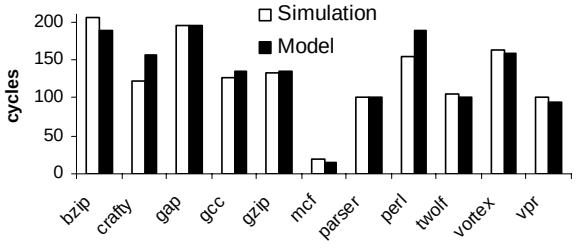

Figure 14: Comparison of penalty per long data cache miss from simulation and model.

5. Evaluation of the First-Order Model

We have now completed all the components of the first-order superscalar model. To demonstrate its accuracy, we evaluate the components of the model and overall performance as follows:

1) Using IW characteristic, average functional unit latency and Little's Law we compute the steady-state IPC (as explained in Section 3).

2) Model the branch misprediction penalty as the average of 5 and 10 cycles (i.e. 7.5 cycles) as in Section 4.1.

3) Model the L1 instruction cache miss penalty as 8 cycles as described in Section 4.2; and L2 miss penalty as 200 cycles.

4) Model the long data cache miss penalty as calculated in equation(8) taking *isolated_dcache_misspenalty* as 200 cycles.

5) Use trace-driven simulations to arrive at the numbers of branch mispredictions, instruction cache misses, data cache misses, and distributions of the bursts of long data cache misses that occur within *rob_size* instructions of a previous long data cache miss.

6) Compute ideal CPI and CPI loss due each type of miss event. Then the CPIs are added as in equation(1) to get the overall CPI. We do not compensate for branch mispredictions and i-cache misses that are overlapped by a d-cache miss. As shown by our initial simulation experiment, these overlaps seem to be only a second-order effect, and will be accounted for in future research.

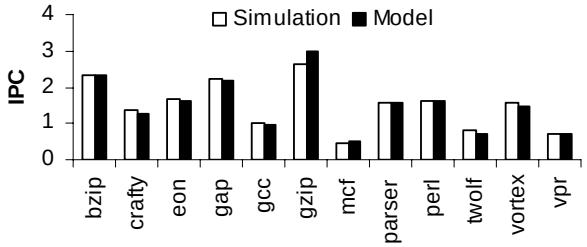

Figure 15: Comparison of performance predicted by our first-order model and from simulation.

Figure 15 gives performance as estimated by the superscalar model compared with detailed, clock cycle level simulations. There is very close agreement between the simulation and the model; the average CPI error is 5.8%. *Mcf, gzip,* and *twolf* are the benchmarks with high errors of 13%, 12% and 12%, respectively.

Because delays independently add, we can build a "stack model" of performance in Figure 16. We observe that all three of the miss-events are important, although their relative importance varies across the benchmarks. We observe that for *mcf* and *twolf*, performance loss because of long data cache misses accounts for 70% and 60%, respectively, of their overall CPI, and these are the two benchmarks with high branch mispredictions. As mentioned earlier we have not taken into account the overlaps of other miss-events with the long data cache misses. Looking at *gzip*, most of the performance loss is due to branch mispredictions. The penalty we use for the first-order model is 7.5 cycles for every branch misprediction, but the penalty measured through simulation is 10 cycles. This indicates the need for greater accuracy in modeling these particular miss-events; i.e. the method of taking a simple average can be improved upon.

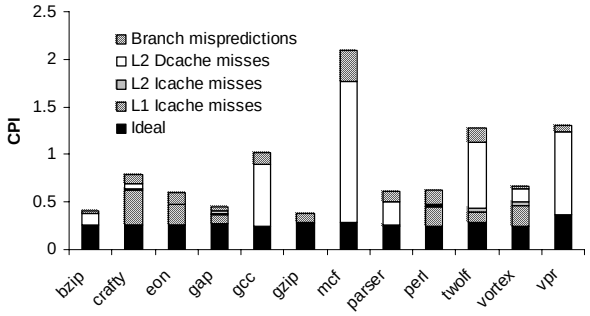

Figure 16: "Stack model" of performance showing the CPI contributions of different miss-events.

6. Application: Trends in Microarchitecture

In this section we demonstrate usefulness of the superscalar model by briefly considering trends in superscalar microarchitecture. In particular, we look at increasing pipeline depth and increasing issue width.

6.1 Increasing Pipeline Depth

The performance effects of increasing pipeline depth have been modeled extensively. Kunkel and Smith [17] determined an optimal pipeline depth through simulation. More recently, Hartstein and Puzak [3], Sprangle and Carmean [4] and Hrishikesh, et al. [18] studied the effects of pipeline depth on superscalar performance.

We focus on the situation where branch mispredictions are the major limiter to increasing pipeline depth. We assume that one of five instructions is a branch and 5% are mispredicted. Then, we use the superscalar model to compute IPC. Figure 17a is the IPC as a function of front-end pipeline depth for issue widths of 2, 3, 4 and 8. As the front-end pipeline deepens the advantage for wider issue is lost (as would be expected). Next, we convert to absolute performance. We use a total delay for the front-end pipeline as 8200 ps and the flip-flop overhead as 90 ps (numbers taken from [4].). If the front-end pipeline is *n* stages then the clock cycle time is (8200ps/n)+90ps. The performance as a function of pipeline depth is given in Figure 17b. For the issue width 3 curve we get the same result as reported in [4], the optimal pipeline depth is around 55 front-end stages. Observe that the optimal pipeline depth for wider issue-width moves towards shorter front-end pipeline depth; this effect is also observed in [3].

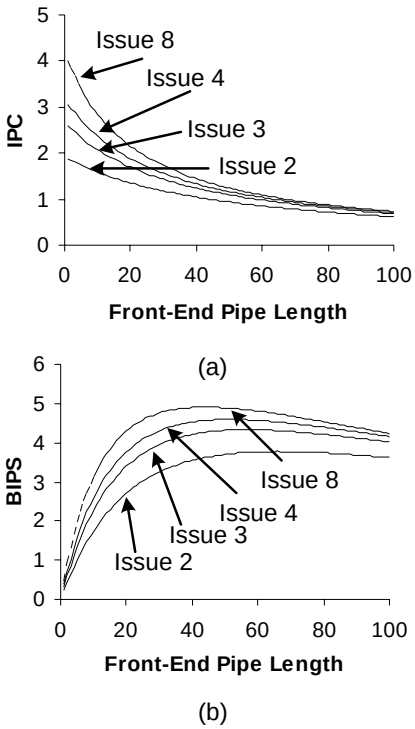

Figure 17: The implication of increasing front-end pipeline length on future pipelines. (a) As the front-end length increases the benefit of wider issue diminishes. (b) Optimal pipeline depths for various issue widths based on the front-end pipeline delay and flip-flop overhead from [4].

6.2 Increasing Issue Width

To study the effects of increased issue width, we again focus primarily on the impact of branch mispredictions. To do this, we consider the fraction of time that the number of useful instructions actually issued is *close* to the implemented maximum issue width. "Close" is defined as anything within 12.5% of the implemented maximum issue width; e.g. if the designed issue width is 8, and the processor model issues 7 during a given cycle, then we count it as achieving the designed issue width during that cycle. Figure 18 shows number of instructions between mispredictions that are required for a given fraction of time that IPC is *close to* the implemented issue width. What this graph shows is that if the same fraction of time is to be spent *close* to the implemented issue width when the issue width is doubled, then the number of instructions between branch mispredictions must quadruple. That is, the performance of the branch predictor (as measured by the number of instructions between mispredictions) must improve as the square of the issue width increase. Given the small incremental gains in branch prediction that are currently being made, this does not bode well for increased issue widths. Of course, deeper pipelines only exacerbate the problem. Figure 19 further illustrates the problem. The graph plots the instructions issued per cycle between an average distance pair of branch mispredictions. The front-end pipeline depth is five. With maximum issue width four, the IPC barely reaches four before a misprediction occurs. With issue width of eight, IPC barely gets above six. Michaud et al. [7] observed a similar effect when studying instruction fetch requirements.

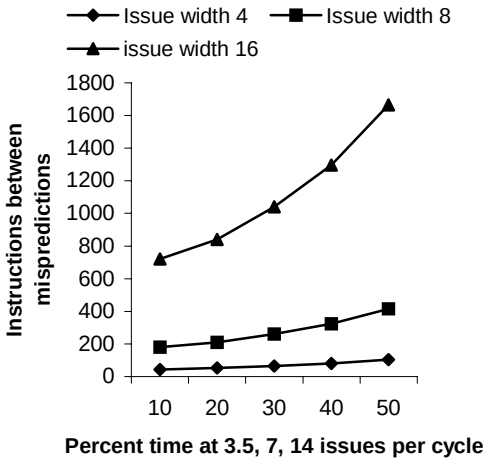

Figure 18: Instructions between two mispredictions as a function of the fraction of time spent within 12.5% of the implemented issue width.

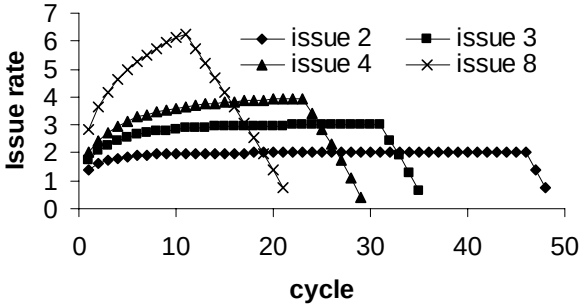

Figure 19: Per cycle instruction issue rate between two mispredicted branches.

7. Summary, Conclusions, Future Work

We developed a superscalar model where a background IPC level is determined, transient penalties due to miss-events are calculated, and these model components are combined to arrive at accurate performance estimates. Using trace-driven data cache misses, instruction cache misses, and branch misprediction rates, the model can arrive at performance estimates that, on average, are within 5.8% of detailed simulation. Further, it provides

some interesting intuition regarding superscalar processors, for example:

1) The branch misprediction penalty is often significantly larger then the front-end pipeline depth.

2) Instruction cache penalty is independent of the front-end pipeline; it depends largely on the miss delay.

3) The data cache penalty for an isolated long miss is essentially the miss delay. For multiple misses that occur within a number of instructions equal to the ROB size, the combined miss penalty is the same as an isolated miss.

Furthermore, the model can be used to arrive at general conclusions regarding superscalar microarchitecture trends. In a pair of brief analyses:

1) We were able to reproduce optimal pipeline depth results derived previously using clock-cycle simulation-based models [3, 4, 17, 18].

2) We were able to show that branch prediction accuracy must improve as the square of issue width if the same IPC profile is to be maintained.

The first order model works very well, and gives us confidence that even more accurate superscalar processor models can be built. There are two avenues for future research: one is to refine the modeling of the features currently in the first-order model, the other is to model new features.

With respect to better modeling of the features currently in the model, we consider the following to be among the more important.

1) Improve modeling of the IW characteristic. The IW characteristic is important for determining the sustained "background" performance level and for determining performance penalties for miss-events.

2) Improve modeling of data cache miss overlap effects. For overlapping long data cache misses, the approximation we used in Section 4.3 is fairly rough and is a weak link in the model, as shown by the errors in *mcf* and *twolf*. We expect that these overlaps can be modeled more accurately (possibly at the cost of more detailed overlap statistics). Also modeling overlaps of instruction cache misses and branch mispredictions with long data cache misses will likely improve accuracy somewhat.

3) Modeling bursts of branch mispredictions. The error in *gzip* suggests that the effect of bursts of branch mispredictions needs to be modeled more accurately. Branch mispredictions are the one case where the window drain penalty and ramp-up penalty do not offset each other. Bursts of branch mispredictions can have significantly less overall penalty than isolated ones. Here, we can collect secondary branch misprediction statistics to better model bursty behavior.

With respect to new features, there are many possibilities; currently, we planning the following:

1) Limited numbers of functional units. Here, we will have to collect instruction mix statistics. To sustain the estimated sustained performance, the mix can be used to determine the number of units required to meet this performance. Or, if the number of units is too small, we can generate a lower saturation level than the maximum issue width. Here, it may be necessary to consider program phases, and model each of them separately – something we have not had to do thus far.

2) Instruction fetch buffers. These buffers immediately follow the instruction cache and can hide some (or all) of the I-cache miss penalty. A related issue is the accounting for fetch inefficiencies due to branching into the middle of cache lines.

3) Partitioned issue windows and clustered functional units. Many superscalar processors have multiple issue buffers, divided according to function or according to sets of clustered units.

4) Additional types of miss-events, TLB misses in particular. When added, these will act much like long data cache misses.

Finally, although they have apparently fallen into disrepute in recent years, we have shown that simple trace driven simulations of caches and branch predictors have a definite, useful role to play in performance evaluation. Cache and predictor miss-events can be directly related to overall performance losses in a fairly straightforward way.

8. Acknowledgements

We thank Timothy Heil for his helpful suggestions. Comments from anonymous reviewers are also appreciated. This work is being supported by NSF grant CCR-0311361, IBM and Intel.

9. References

[1] G. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," *International Symposium on Computer Architecture*, pp. 27-34, 1987.

[2] P. G. Emma and E. S. Davidson, "Characterization of Branch and Data Dependencies on Programs for Evaluating Pipeline performance," *IEEE Transactions on Computers*, vol. 36, pp. 859-875, 1987.

[3] A. Hartstein and T. R. Puzak, "The Optimum Pipeline Depth for a Microprocessor," *International Symposium on Computer Architecture*, pp. 7-13, 2002.

[4] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *International Symposium on Computer Architecture*, pp. 25-34, 2002.

[5] D. B. Noonburg and J. P. Shen, "Theoretical Modeling of Superscalar Processor Performance," *Interna-*

tional Symposium on Microarchitecture, pp. 52-62, 1994.

[6] P. Michaud, A. Seznec, and S. Jourdan, "Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors," *International Symposium on Parallel Architectures and Compilation Techniques*, 1999.

[7] P. Michaud, A. Seznec, and S. Jourdan, "An Exploration of Instruction Fetch Requirement in Out-Of-Order Superscalar Processors," *International Journal of Parallel Programming*, vol. 29, 2001.

[8] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *International Symposium on Parallel Architectures and Compilation Techniques*, 2001.

[9] R. Carl and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Workshop on Performance Analysis and Its Impact on Design*, 1998.

[10] L. Eeckhout, K. De Bosschere, and H. Neefs, "Performance Analysis Through Synthetic Trace Generation," *International Symposium on Performance Analysis of Systems and Software*, 2000.

[11] D. B. Noonburg and J. P. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance," *International Symposium on High Performance Computer Architecture*, pp. 298-309, 1997.

[12] D. Sorin, V. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic Evaluation of Shared Memory Systems with ILP Processors," *International Symposium on Computer Architecture*, pp. 380-391, 1998.

[13] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Using Interaction Costs for Microarchitectural Bottleneck Analysis," *International Symposium on Microarchitecture*, pp. 228-239, 2003.

[14] D. J. Ofelt, "Efficient Performance Prediction for Modern Microprocessors," Stanford University PhD Thesis, 1999.

[15] E. Riseman and C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, pp. 1405-1411, 1972.

[16] N. P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Transactions on Computers*, vol. 38, pp. 1645-1658, 1989.

[17] S. R. Kunkel and J. E. Smith, "Optimal pipelining in supercomputers," *International Symposium on Computer Architecture*, pp. 404-411, 1986.

[18] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays," *International Symposium on Computer Architecture*, pp. 14-24, 2002.