RC 20962 (92834) 08/28/1997
Computer Sciences/Mathematics

IBM Research Report

Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors

Jaime H. Moreno, Mayan Moudgill, John-David Wellman,
Pradip Bose, Louise Trevillyan
{jmoreno, mayan, wellman, bose, louise}@watson.ibm.com,

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

NON-CIRCULATING

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Research Division
Almaden • Austin • China • Haifa • Tokyo • T.J. Watson • Zurich

Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors

Jaime H. Moreno, Mayan Moudgill, John-David Wellman,
Pradip Bose, Louise Trevillyan
{jmoreno | mayan | wellman | bose | louise}@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

We investigate the potential performance of PowerPC-based wide superscalar processors on a standard on-line transaction processing (OLTP) workload, using an instruction and data reference trace as input. We explore instruction-level parallelism as a function of the width of the processor, branch prediction accuracy, size of cache memory, and policy for issuing instructions for execution. Performance factors are identified from the perspective of instruction retirement, highlighting the degradation contributed by different sources. The sensitivity to selected microarchitecture features is also studied. Simulation results validate common wisdom regarding the degrading effects of the memory subsystem, regardless of the width of the processor. The data also show that, for the workload and processor organizations considered, (1) dispatching more than eight instructions per cycle provides diminishing performance improvement; and (2) performance gains arising from out-of-order instruction issue increase with the processor width.

1. Introduction

Contemporary high-performance general-purpose microprocessors are classified as *superscalar* engines because of their ability to process more than one instruction at a time. Many such processors issue instructions out-of-order, that is, in an order that differs from the order in which instructions appear in the program. Moreover, most superscalar processors also support speculative execution, wherein instructions that appear after a conditional branch operation (either in the fall-through path or in the branch target path) are issued for execution before the branch is resolved, based on a prediction of the branch outcome [1].

An important field of contemporary computing corresponds to data-intensive applications, such as transaction processing, file servers, and data mining. Collectively, these are referred to as *commercial* applications because they were originally used mostly by commercial enterprises. Several standard benchmarks have been developed in this field, with the objective of serving as good indicators of system performance for specific segments of the commercial market. For example, the TPC-C benchmark is intended to simulate an order-entry environment with a mix of read-only and update-intensive transactions [2].

Processor features such as those described above make microarchitectures increasingly complex. Superscalar, out-of-order instruction issue and speculation are only some of a large domain of design options. This complexity raises questions regarding the potential performance benefits obtained from the features. Since the interaction among microarchitecture features is often counterintuitive, early, accurate, and timely modeling are required to ensure proper design trade-offs [3-4]. In other words, the benefits of various microarchitecture features should be quantified properly so that those leading to effective and efficient implementations can be identified [5]. Moreover, the benefits of such features differ among the various areas of application; for example, the characteristics of commercial applications differ dramatically from those of numeric-intensive or CPU-intensive ones, making it necessary to understand the effects of commercial workloads on processor performance [6-7].

Much work is constantly reported on the evaluation of microarchitecture features. Usually, such reports study the benefits of a specific new proposal instead of evaluating the impact of a variety of features. Moreover, most of the literature addresses the impact of microarchitecture features on numeric-intensive or CPU-intensive benchmarks, such as the SPEC suite [8]. Few reported studies address the impact on commercial applications [6-7], among other reasons due to the difficulty in obtaining suitable test inputs (in terms of size and representativeness) that can be use for such investigations. As a result, there is a need for data from the systematic exploration of microarchitecture features with commercial workloads; results from such an exploration will be helpful in

understanding the processor characteristics most adequate for that area.

In this report, we investigate the potential performance of PowerPC-based wide superscalar processors on a standard on-line transaction processing (OLTP) benchmark, using a PowerPC 601 instruction and data reference trace as the workload. We first summarize the characteristics of our exploration approach, then describe the features of the processor model used in the exploration, followed by the configurations explored and the results from the experimentation. Such results include the average cycles per instruction (CPI) obtained in each configuration, details on degradation factors that limit performance, and sensitivity to some selected microarchitecture features.

The simulation results reported here show that delays due to misses encountered in caches and translation look-aside buffers are an important limiting factor, thereby validating common wisdom regarding the degrading effects of the memory subsystem on this workload; such behavior is encountered regardless of the width of the processor.

Furthermore, for the trace and processor organizations considered, the simulation data show that increasing the processor dispatch width to eight operations per cycle is advantageous whereas wider organizations provide diminishing performance improvement. For example, doubling the features of an out-of-order processor whose dispatch width is four by doubling the number of units and the various widths, while preserving the size of caches and prediction accuracy, produces about 20% overall performance improvement. Doubling also the size of the caches in the same configuration produces an additional 10% improvement.

In addition, the simulation results also indicate that wider processors achieve larger performance gains when instructions are issued out-of-order. For example, doubling the features of a processor as in the paragraph above increases the performance advantage of out-of-order issuing of instructions over in-order from 15 to 30%.

2. Exploration methodology

Our objective is the development of an understanding of limits and potentials of highly-concurrent superscalar processors on a standard OLTP benchmark. The experiments described in this report do not attempt to obtain accurate results for any specific implementation, but instead determine performance trends among some of the many variables involved. Furthermore, the processor model used in the exploration makes some assumptions or implements certain features which could be regarded as too aggressive or too conservative for current or next-generation technology, depending on the specific point-of-view. Moreover, we do not address issues such as clock cycle time or variations in the processor organization (other than number of pipeline stages, number of units, and so on), and assume them constant for all the configurations. Consequently, the importance of the results obtained in this study is not the magnitude of the values but the trends among those values.

As many other efforts towards studying/predicting the performance of new microprocessors, this exploration uses trace-driven simulation [3][6]. The standard OLTP workload is represented by an *execution trace* which contains 172 million instructions plus data addresses, and includes instructions executed in user and kernel address spaces. Other features of the input trace are summarized in Table 1. The validity of this trace as a representative characterization of the original workload is analyzed in [9].

Table 1: Characteristics of input trace

% of branch instructions	18.9
% of branches taken	46.3
% of instructions in kernel space	23.6
% of memory access instructions	34.8
% of load/store multiple instructions	1.6
% of string instructions	1.4
% of load/store with update instructions	1.7
average block size (instructions.)	5.3

We explore instruction-level parallelism as a function of

- the policy for issuing instructions;
- the width of the processor;
- the accuracy of the branch predictor; and
- the size of a two-level cache memory.

We construct 48 configurations within the exploration space defined by these orthogonal dimensions, ranging from idealized values that illustrate the extreme of some dimension (such as infinite cache or perfect branch prediction), to values that appear achievable with current or next generation technology (such as 64KB first level caches and 2MB second level). In addition, we explore the sensitiveness of selected configurations to some microarchitecture features.

We use *cycles per instruction (CPI)* as the performance metric, which corresponds to the *average* number of processor cycles required to execute an instruction. Consequently, a lower CPI value represents better performance than a higher value. The meaningfulness of using CPI as a performance metric, as opposed to its inverse *instructions-per-cycle* (IPC), is discussed in [10]. The *minimal CPI* for a given processor organization is the inverse of the

peak IPC attainable; such peak value is usually obtained from the maximum number of instructions which can be retired in one cycle. The *minimal sustained CPI* over the course of execution of a program is determined by the *minimum IPC bandwidth* in the overall processor organization. For example, if the processor *dispatch bandwidth* is 4 instructions per cycle, and the *retirement bandwidth* is 5, the best case sustained IPC is 4, not 5, even though the peak IPC achievable in an instantaneous sense is 5.

Out-of-order superscalar processors include the ability to fetch and dispatch operations from a predicted path before branches are resolved, thereby executing those operations speculatively. Upon branch resolution and detection of branch misprediction, the speculative operations must be cancelled and execution resumed from the correct target address of the branch. However, there are cases in which the effects of the speculative operations may not be cancelled, such as a load operation which already replaced a block in the cache, or a long latency operation which has already been issued to a functional unit. Modeling these effects requires the availability of the instructions from the mispredicted path and the memory addresses of the locations accessed by those instructions; however, this information is not present in the trace used as input in this study. As a result, this exploration does not include the effects arising from those instructions. However, evidence collected elsewhere [11] suggests that the effects of speculative operations are not necessarily negative, since speculation may act as a source of prefetching.

We assume that operations are classified according to the functional unit where they are executed, as follows: integer, memory, floating-point, and branches. Moreover, we assume that there is an *issue queue* for each class of operations. Operations are dispatched into the corresponding issue queues in the order in which they appear in the program (*in-order dispatch*). Two policies for removing operations from the issue queues and issuing them for execution are explored:

- out-of-order; and
- class-order.

The *out-of-order* policy issues each operation for execution as soon as the required operands and functional unit are available, regardless of the order in which the operations were inserted in the issue queues. On the other hand, the *class-order* policy issues the operations belonging to the same class of functional unit in program order, but allows out-of-order issuing among operations belonging to different classes. That is, operations in the same class are issued for execution in the order in which they were inserted into the issue queue, but there is no ordering imposed with respect to operations in other issue queues. Note that this is different from an *in-order* policy, wherein ordering is enforced across all classes of functional units (i.e., in-order dispatch into a single issue queue for all units, in-order issue from this queue). Since class-order issuing is less restrictive than in-order issuing, class-order issuing can be regarded as an upper bound in performance for in-order issuing.[*]

As already stated, the experiments are trace-driven; no prior initialization of the state of the processor is assumed (e.g., caches and queues are assumed empty).

The experiments have been carried out using a set of tools developed for supporting fast simulation of microprocessor configurations [12], so that multiple microarchitecture variations can be explored promptly. These tools permit throughput in excess of 100 million simulated processor cycles per hour on a contemporary workstation (such as a RS/6000 43P model 140, which has a PowerPC 604e processor running at 166Mhz [13]).

3. Processor model

The exploration of instruction-level parallelism reported here uses a superscalar processor model which is extensively parameterized, so that it can cover a large variety of microarchitecture features [14]. Parameters include changing the size of the various resources, the number and latency of functional units, the number of pipeline stages, enabling/disabling features, and so on.

An overall view of the processor model is depicted in Figure 1. It consists of the following elements:

- a memory subsystem with the following components:[†]
  - separate 2-way set-associative first level instruction and data translation look-aside buffers (I-TLB and D-TLB, respectively);
  - 2-way set-associative second level TLB (TLB2);
  - separate 4-way set-associative first level instruction and data caches (L1-I and L1-D, respectively);
  - 2-way set associative second level cache (L2);
  - data cache miss queue;
  - store queue;
  - cast-out queue;
  - L1-L2 bus; and
  - main memory.
- a load/store reorder buffer;
- an instruction prefetch unit with an associated instruction prefetch buffer, which anticipates the demands for blocks of instructions accessed sequentially so that L1-I cache misses may be served faster than accessing the L2 cache;
- a next-fetch-address/branch-prediction unit (NFA/BP), which every cycle predicts (1) the address of the

[*] The superscalar processor model used in this study does not implement in-order issuing of instructions.

[†] The size, line size, latency and miss penalty of these structures are controlled by parameters; however, the associativity of the structures cannot be changed.

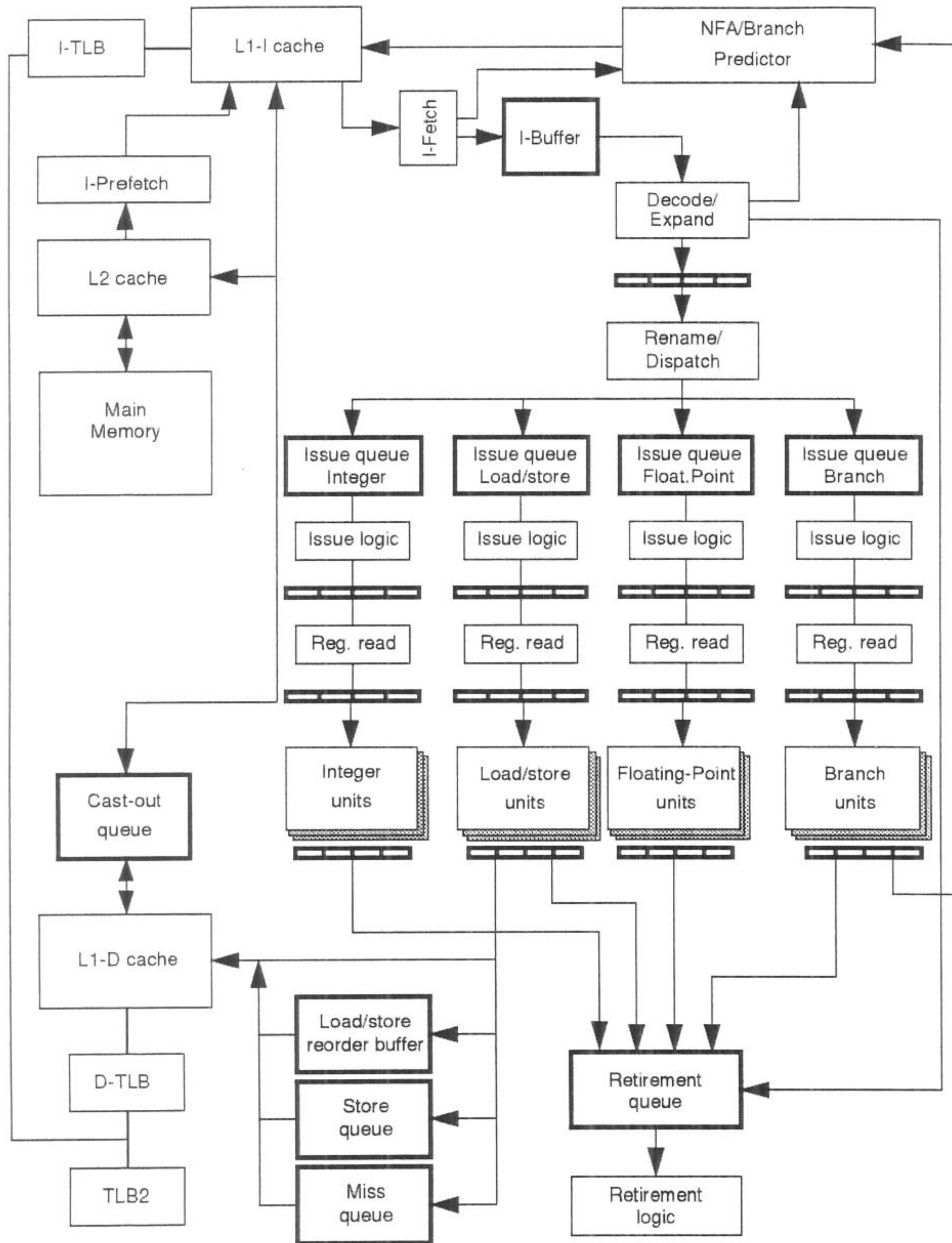


Figure 1: Processor organization

next group of instructions to be fetched, and (2) the outcome of conditional branches;

- an instruction fetch unit (I-fetch), which fetches instructions into the instruction buffer (I-buffer);
- issue queues for each class of functional unit, which hold dispatched operations that are waiting either for availability of operands and/or functional unit;
- separate multiported register files for general-purpose registers, floating-point registers, condition code register fields, and special-purpose registers;
- a collection of functional units, classified as integer units, memory (load/store) units, floating-point units, and branch units; each unit has a separate path to place its result in the corresponding destination;
- retirement queue, which holds all the operations that have been dispatched (and perhaps even executed) but which have not yet been completed;
- logic to decode/expand, rename/dispatch, issue, and retire instructions, distributed across several pipeline stages.

The processor model implements a conventional pipeline, as depicted in Figure 2. Integer operations, including branches, traverse the pipeline in eight cycles (stages), whereas memory operations and floating-point operations require ten cycles. Branches are subject to early resolution at dispatch time, as described later, but they still traverse the pipeline as indicated in the figure. Additional cycles are required in the case of data cache misses and long-latency operations (such as divide, square-root, and so on).

Instruction prefetch. Instruction prefetch uses a *next-sequential* algorithm, as follows: requests for instructions are presented to the instruction cache (L1-I) and instruction prefetch unit simultaneously. If the requested group of instructions is not in the L1-I cache but is present in the prefetch buffer, the group of instructions is transferred to the processor and the corresponding line is transferred into the instruction cache; a prefetch request for the next sequential cache line is sent to the L2 cache. On the other hand, if the requested group of instructions is neither in the L1 instruction cache nor in the prefetch buffer, the corresponding cache line is transferred from the L2 cache into the instruction cache, and the next sequential line is transferred into the prefetch buffer. If a line to be prefetched is not already in the L2 cache, the prefetch request is ignored.

Next fetch-address and branch prediction. A next fetch-address and branch prediction unit (NFA/BP, see Figure 3) is used to generate a new address for accessing the instruction cache in every cycle, and for predicting the outcome of the conditional branches present in the group of instructions fetched in a given cycle.

The branch prediction logic uses a branch history table (BHT) of 2-bit saturating up/down counters [15]. Multiple ports in this table are used for the simultaneous prediction of all branch instructions fetched together.

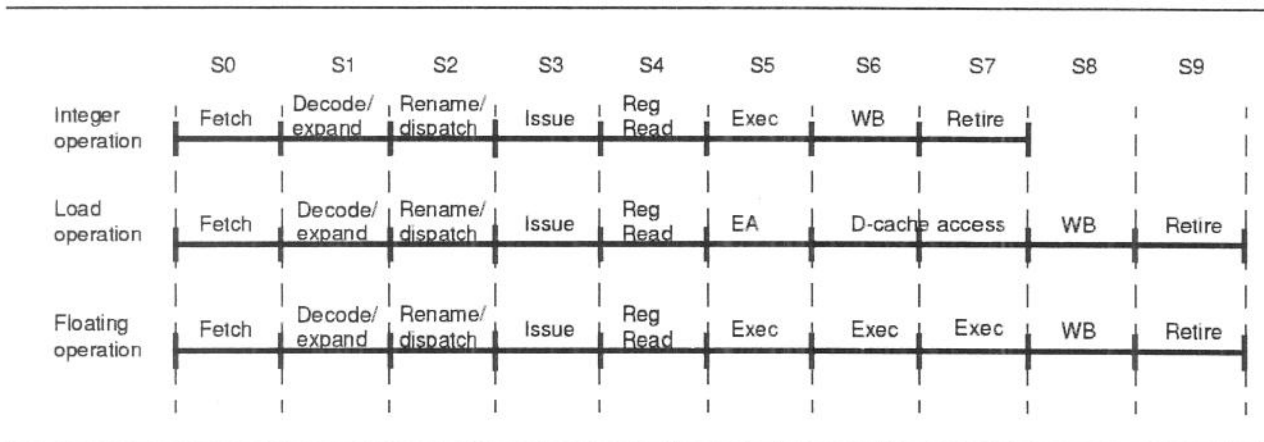


Figure 2: Pipeline stages

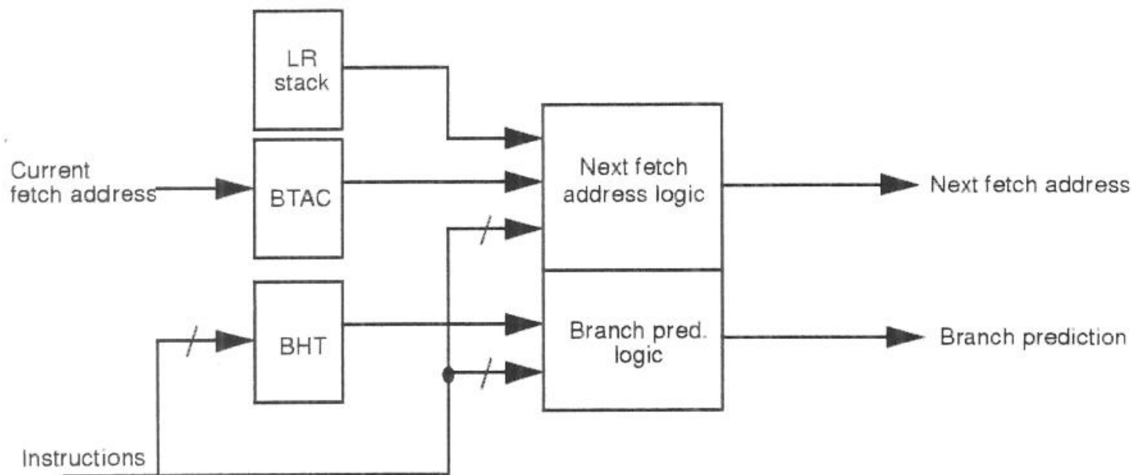


Figure 3: Next fetch address (NFA) and branch prediction units

The next-fetch-address logic chooses among the output from a branch target address cache (BTAC) and the top of a Link Register stack, as follows: if the group of instructions just fetched contains a Branch to Link Register instruction, and all previous branch instruction within the group are predicted as not-taken, then the top of the Link Register stack is used; otherwise, the output from the BTAC is used.*

The BTAC is four-way set associative, and contains only the addresses of branch instructions actually taken; if a branch is not found in the BTAC, it is regarded as a branch not taken so the BTAC output is the address of the block of instructions plus the size of the block. The BTAC is updated whenever there is a mismatch between the address predicted by the branch prediction logic and the address generated by the next fetch-address logic. There is a two-cycle delay from the generation of the next fetch-address to the detection of the mismatch and BTAC update, in which case the prefetched instructions must be discarded and the prefetch started from the address generated by the branch prediction logic (a two cycle pipeline bubble).

The BTAC is indexed by the address generated by the next fetch-address logic in the previous cycle. The next fetch address logic can be forced to operate under a "perfect prediction" mode, in which case the address generated is the same address produced the branch prediction logic.

The Link Register stack is updated by all branch instructions which have the *link bit* set; whenever such an instruction is encountered, the address of the next instruction is pushed into the stack. When a branch to link register instruction is encountered, the value at the top of the stack is removed and used as the predicted target.

A mispredicted branch leads to stalling the pipeline until the branch is resolved, and resuming execution from the target of such branch. Stalling is used as a substitute for pipeline flushing of operations executed speculatively (which cannot be simulated because those operations are not present in the input trace).†

Instruction fetch. Instructions are transferred into the instruction buffer starting from the address specified by the NFA/BP unit. The number of instructions fetched in a given cycle equals the fetch width of the processor only if the starting address is aligned at a fetch width boundary (a *fetch block*). Otherwise, instructions are fetched up to the end of the fetch block (i.e., fetching does not straddle fetch block boundaries neither cache line boundaries).

Instruction expansion. Complex PowerPC instructions (such as load/store-multiple or string instructions), as well as load-with-update and store instructions, are decomposed in the decode/expand stage into multiple *primitive operations*. As a consequence, the total number of operations flowing through the processor pipeline beyond the decode/expand stage is larger than the number of instructions in the input. All queues and buffers other than the I-buffer contain primitive operations

* Some of these features might not be realizable in practice, due to cycle time and/or hardware limitations.

† The processor model actually has the capability to take into account the effect of speculatively executed operations if they are present in the input trace [14]. This capability can be exercised by using the micro-trace generation engine described in [16].

(instead of PowerPC instructions); this includes the issue queues as well as the retirement queue. Multiple primitive operations are generated per cycle, up to the decode width; consequently, multiple cycles may be required to decompose an instruction.

In the rest of this report, we use the term *instruction* to refer to the PowerPC instructions fetched from memory, and the term *primitive operation* or just *operation* to refer to the operations flowing through the pipeline after the decode/expand stage.

Store instructions are expanded into an address generation (*agen*) operation and a data move (*dmove*) operation. Store with update instructions are expanded into operations *agen*, *update* (i.e., an add) and *dmove*. Load with update instructions are expanded into a *load* and an *update* operation.

Complex memory instructions which explicitly indicate the number of memory transfers required, such as *string immediate* or *load/store multiple* instructions, are decomposed into a corresponding number of primitive operations. In contrast, complex instructions whose number of memory transfers is run-time dependent, such as *string indexed* instructions, are decomposed into the maximum possible number of memory transfers (worst-case condition); the resulting primitive operations include the detection of whether each of the memory transfers should be performed or not, by checking the value of the register that determines the number of such transfers. For example, instruction *lswx* is decomposed into 96 primitive operations, so that the decode stage is used by this instruction during 24 cycles.

The expansion ratio observed in the trace under study is 2.01, so that 346 million primitive operations are retired. The results presented later regarding utilization of processor resources are given in terms of primitive operations; however, performance results correspond to *cycles per PowerPC (non-expanded) instruction*.

Register renaming. Register from the different classes are renamed separately. This includes general-purpose registers, floating-point registers, condition register fields, and special-purpose registers (such as XER fields). Each class has its own physical register file, as well as separate structures for keeping track of free/used registers.

Load/store reorder buffer and store queue. At the rename/dispatch stage, the load/store reorder buffer is checked for availability of entries for load operations. Similarly, the store queue is checked for entries for *agen/dmove* operations arising from store instructions. That is, load and *dmove* operations are not dispatched unless there is an entry available in the corresponding queue. If such an entry is not available, the rename/dispatch stage is stalled until one becomes available.

Operation issuing. Operations are issued for execution one cycle before the corresponding functional unit is available and all the required operands are ready. Operands already available are read from the register file in the cycle after the operation was issued, whereas operands which become ready the next cycle are received through the bypass mechanism by the time the operation reaches the functional unit (i.e., one cycle later). In the case of multiple operations available for issuing, they are selected according to the order in which they were inserted into the issue queue (i.e., oldest first).

Operation execution. Operations issued to integer, branch and floating-point units complete execution within a fixed number of cycles. In contrast, memory operations complete execution in a variable number of cycles, due to the varying latency of memory operations and the contention for the resources (queues, ports) to access memory.

Memory operations. A memory operation that generates a second level TLB miss stalls *all* the memory units for a specified number of cycles. Similarly, a memory operation that does not find an entry available in the miss queue or in the cast-out queue stalls the corresponding memory unit. Cache misses lead to placing the memory operation in the miss queue. A *dmove* operation marks the corresponding entry in the store queue as having data available.

Writing results. Each unit has a separate write port to the corresponding register file, so that results are written in the register file without contention. In contrast, data cache ports are shared among the memory units and the cache replacement logic.

Data cache ports. Data cache ports are accessed in the following order:

1. cache line replacement uses two ports: one for writing the new data, and the other for (potentially) reading the dirty line being replaced;
2. each load operation uses one port; and
3. each *dmove* operation (generated from a store instruction) uses one port.

Retiring operations. Operations are retired in program order after they have placed their results in the corresponding destinations. In the case of *dmove* operations arising from store instructions, they are retired when the data has been sent from the store queue to the data cache (i.e., when there is a cache port available).

Validation of the model

An important aspect in the simulation and evaluation of processor microarchitectures is the correctness of the model used. Given the complexity of the microarchitec-

ture features being simulated, and the non-intuitive interaction among those features, it is necessary to perform suitable validation tests.

We carried out extensive testing of the model, applying techniques similar to those used in testing and validation of more detailed models and actual processors [4]. This included the generation of test vectors with expected behavior and cycle count, and the verification of their correct execution. Additional details regarding this validation procedure are described in [17].

In addition, we performed extensive cross-checking among the data generated by the model. Since data are collected at different places within the model, we used that data to verify consistency throughout the various pipeline stages and units.

4. Processor configurations evaluated

The processor configurations used in this study explore four dimensions, as follows:

- **Policy for issuing instructions:** *out-of-order* or *class-order*.
- **Fetch/dispatch/retire width:** 4/4/6, 8/8/12 or 12/12/16 instructions per cycle.
- **Size of a two-level cache memory:** 64KB first level and 2MB second level, to infinite size.
- **Accuracy of the branch predictor:** history table with 8192 saturating 2-bit counters [15], or perfect branch prediction.

The values of the parameters for the configurations explored are summarized in Table 2, wherein the character sequence (in boldface) to the left of each parameter value is used to compose the names identifying the configurations in the results presented later. For example, a configuration named *c4StPf* corresponds to a processor with class-order issuing of operations, fetch/dispatch width equal to 4, 64K/64K/2M cache memory, and perfect branch prediction. The combination of all these parameter values leads to the 48 configurations explored. The number of functional units, size of certain queues and parameters that depends on the fetch/dispatch/retire width of the processor are listed in Table 3.

Features of the processor which were kept the same for all configurations are listed in Table 4 (the description of these and other parameters is given in [14]).

Table 2: Exploration dimensions

Issue policy		Width: Fetch, Disp, Retire		Cache size: L1-I, L1-D, L2		Branch prediction	
c	Class-order	4	4, 4, 6	**St**	64K, 64K, 2M	**Bp**	2048 2-bit counters
o	Out-of-order	8	8, 8, 12	**Lg**	128K, 128K, 4M	**Pf***	Perfect
		12	12, 12, 16	**IL2**	128K, 128K, Inf		
				Inf†	Inf, Inf, Inf		

* Perfect branch prediction also implies perfect next fetch address prediction

† Infinite cache also implies infinite TLBs

Table 3: Varying parameters for the various configurations

Widths		Units				Ports		Queues			Pred. branch	Phys. Regs.
Fetch/ dispatch	Retire	FX	FP	LS	BR	Cache	TLB	Issue	Retire	IBuf*		GPR, FPR, CCR, SPR
4/4	6	3	2	2	2	2	2	20, 12†	128	24	12	80, 80, 32, 64
8/8	12	6	4	4	4	4	4	40	160	48	24	128, 128, 64, 96
12/12	16	8	4	6	4	6	6	60	160	72	24	128, 128, 64, 96

* before instruction expansion

† branch issue queue

Table 4: Other important parameters

Maximum instructions in flight	160	Size of miss and cast-out queues (entries)	8
I-prefetch latency (cycles)	1	Size of store queue and load/store reorder buffer (entries)	31
I-prefetch buffer size (cache lines)	4	Cast-out overhead (cycles)	5
I-prefetch latency at prefetch hit (cycles)	8	D/I-TLBs size (entries)	128
I-prefetch latency at L1-miss/L2-hit, after L1 reload (cycles)	4	D/I-TLBs miss penalty (cycles)	4
BTAC size (entries)	4096	TLB2 size (entries)	1024
Next fetch address misprediction penalty	2	TLB2 miss penalty (cycles)	40
LR stack size	32	I, D, L2-cache line size (bytes)	128
Branch history table (entries)	8192	I, D miss penalty (cycles)	8, 7
Page size (bytes)	4096	L2-cache miss penalty (cycles)	40

5. Performance results

We now present the results obtained from this exploration in terms of average cycles per instruction (CPI), and discuss relevant aspects inferred from those results. Table 5 lists the data obtained for all the configurations being evaluated; each entry in the table also contains the name assigned to the configuration, which is used in the charts that follow.

The data from Table 5 is used in Figure 4 to depict CPI as a function of the issue policy. The lower portion of each bar corresponds to the CPI for the case of issuing instructions out-of-order, whereas the top portion of each bar represents the degradation in CPI (*CPI adder*) introduced by class-order issuing of instructions. The numbers inside the bars correspond to the value of the CPI adder expressed as a fraction (%) of the CPI of the corresponding out-of-order configuration. As it can be inferred from this figure, the out-of-order configurations consistently outperform the corresponding class-order configurations (i.e., with same width, cache organization, and branch predictor), the adder being larger for wider configurations, larger caches and better branch prediction. In the case of configurations StBp, the adder ranges from 15 to 32%, whereas in the case of configurations InfPf the adder ranges from 35 to 125%.

Table 5: Cycles per instruction (CPI) for all configurations

Issue policy	Width	2048-entry 2-bit branch predictor				Perfect branch prediction			
		Inf	IL2	Lg	St	Inf	IL2	Lg	St
Class-order	4	c4InfBp 0.82	c4IL2Bp 1.07	c4LgBp 1.18	c4StBp 1.29	c4InfPf 0.72	c4IL2Pf 0.93	c4LgPf 1.03	cgStPf 1.12
	8	c8InfBp 0.71	c8IL2Bp 0.96	c8LgBp 1.07	c8StBp 1.18	c8InfPf 0.62	c8IL2Pf 0.81	c8LgPf 0.91	c8StPf 1.00
	12	c12InfBp 0.70	c12IL2Bp 0.95	c12LgBp 1.06	c12StBp 1.17	c12InfPf 0.60	c12LI2Pf 0.79	c12LgPf 0.89	c12StPf 0.97
Out-of-order	4	o4InfBp 0.67	o4IL2Bp 0.93	o4LgBp 1.02	o4StBp 1.12	o4InfPf 0.53	o4IL2Pf 0.77	o4LgPf 0.86	o4StPf 0.95
	8	o8InfBp 0.44	o8IL2Bp 0.71	o8LgBp 0.81	o8StBp 0.91	o8InfPf 0.31	o8IL2Pf 0.56	o8LgPf 0.65	o8StPf 0.75
	12	o12InfBp 0.41	o12IL2Bp 0.68	o12LgBp 0.77	o12StBp 0.88	o12InfPf 0.27	o12IL2Pf 0.51	o12LgPf 0.60	o12StPf 0.70

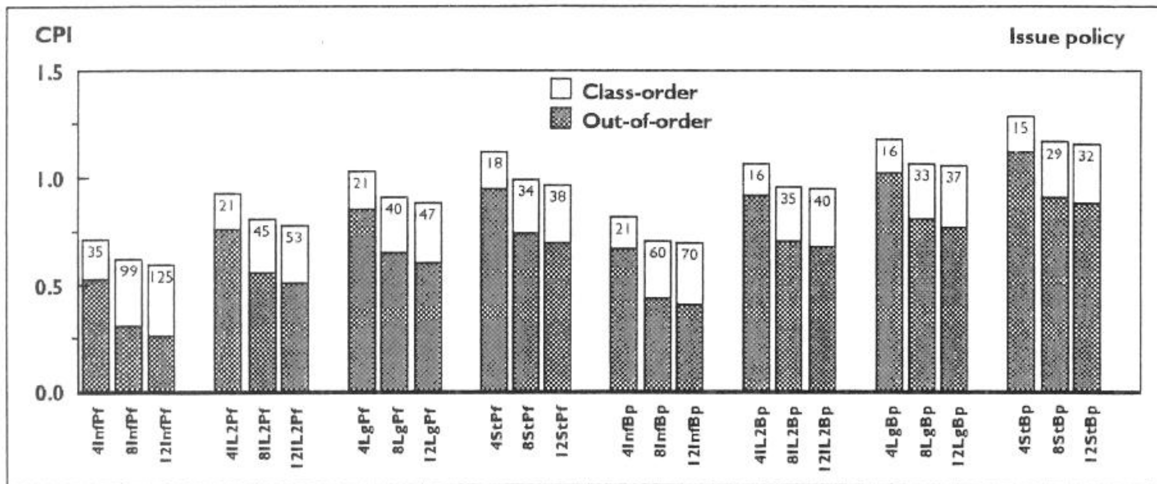


Figure 4: CPI adder due to class-order issuing of operations

Figure 5 uses similar adders to depict CPI as a function of branch prediction. The lower portion of each bar corresponds to perfect branch prediction, whereas the upper portion represents the adder arising from the imperfect predictor. The numbers inside the bars indicate the CPI adder expressed as a fraction (%) of the CPI of the corresponding configuration with perfect branch predictor. For the case of configurations with 64K/2M, this adder ranges from 15 to 26%.

Figure 6 illustrates the effects arising from variations in the features of the cache memory. The lower portion of each bar corresponds to the CPI for the case of infinite cache, which also includes infinite TLB, whereas the upper portions represent the adders due to reducing the size of the cache and the introduction of a finite TLB (the finite TLB is the same in all cases). The numbers inside the bars in the figure indicate the CPI adder (expressed as a per-cent of the infinite cache CPI) introduced when replacing the infinite-cache/infinite-TLB by a 128K-L1/infinite-L2 cache and two-level TLB with 128/1024 entries.

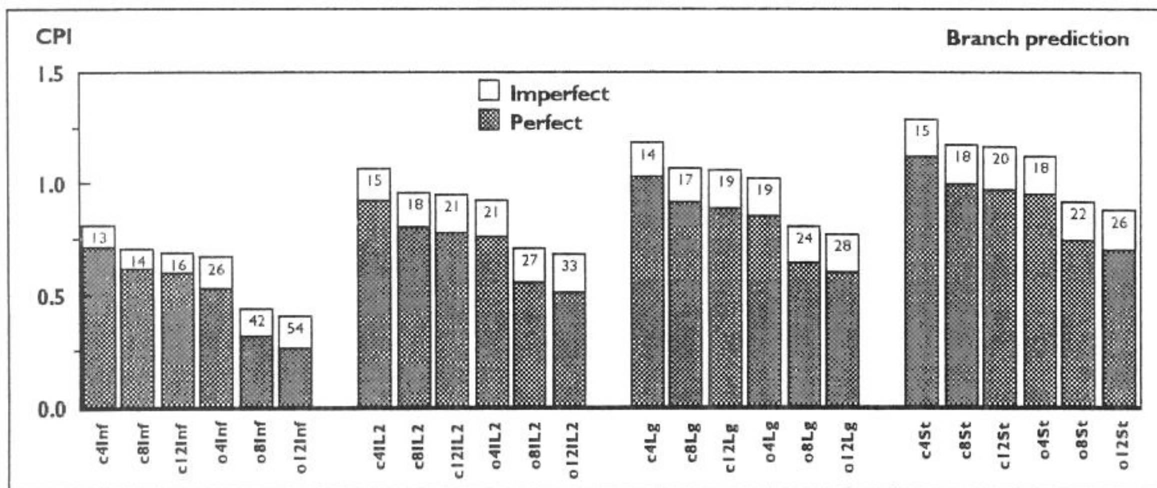


Figure 5: CPI adder due to imperfect branch prediction

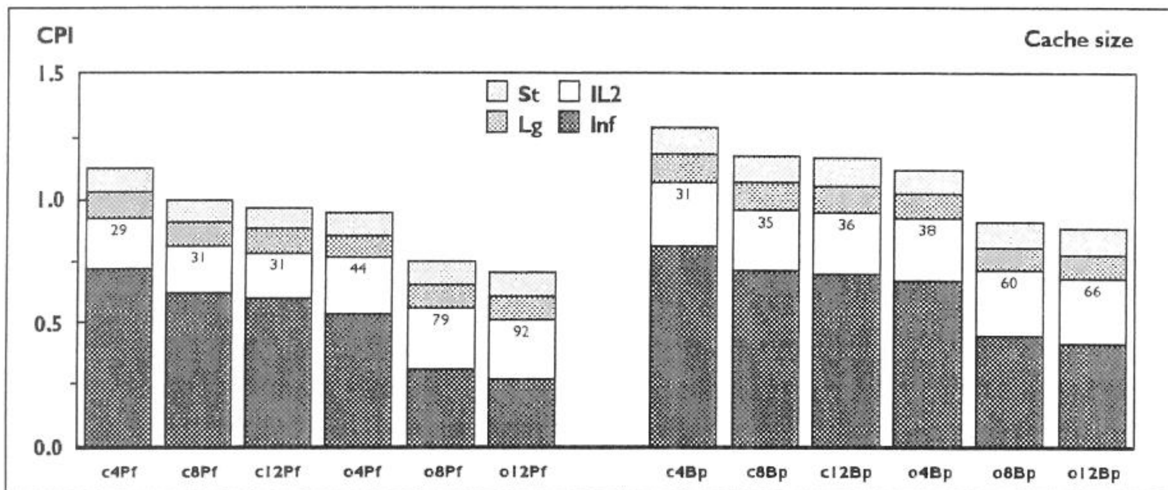


Figure 6: CPI adder due to cache size

Figure 6 shows that any decrease in cache size leads to performance degradation. The CPI adder when the TLB and data cache are made finite ranges from 29 to 92% of the CPI of the corresponding infinite configuration.

Figure 7 depicts CPI as a function of the width of the configurations. In this case, the lower portion of each bar corresponds to the CPI for the widest configuration (w=12), whereas the upper portions represent the adders arising from the narrower widths. The numbers inside the bars correspond to the adder arising from decreasing the width of the processor from 8 to 4 (fetch/dispatch from 8 to 4, retire from 12 to 6, and the number of units is halved); this adder ranges from 10 to 71%. On the other hand, the degradation is rather small when the width is decreased from 12 to 8 (fetch/dispatch width from 12 to 8, retire from 16 to 12, and fewer units). This chart indicates that there are clear advantages in increasing the processor width to fetch/dispatch 8 operations per cycle but not beyond, and such advantages are larger in out-of-order processors. This behavior is found regardless of the cache size and branch prediction accuracy.

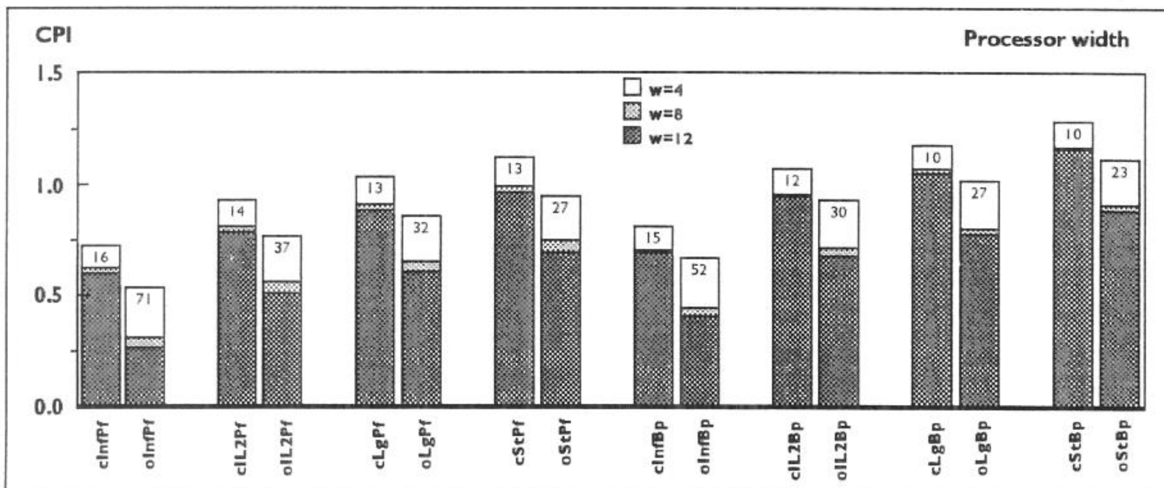


Figure 7: CPI adder due to narrower width

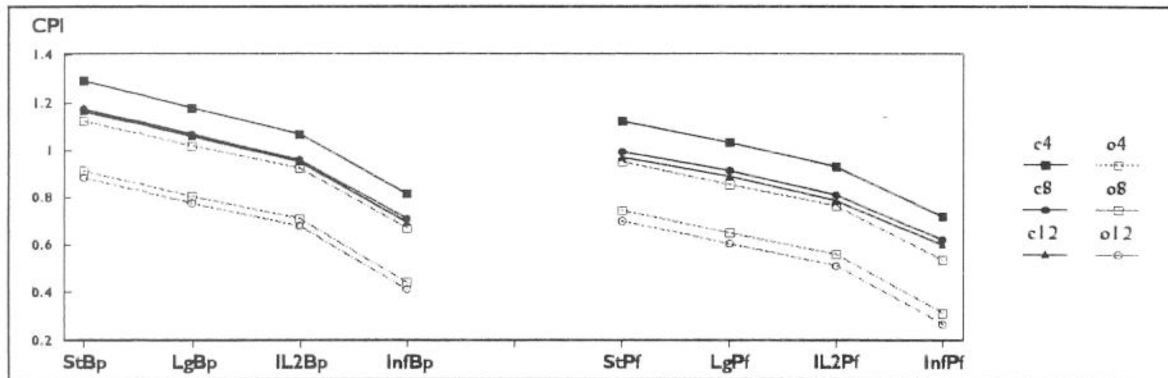


Figure 8: Cycles per instruction (CPI) for all configurations

The diminishing performance improvement seen in Figure 7 could be attributable to two different sources:

- the instruction-level parallelism present in the instruction window under consideration is practically exhausted with a fetch/dispatch width of 8; or
- some of the fixed resources (queues, buffers) in the processor saturates.

Using the detailed data gathered by our simulator, which is described later, we verified that queues are not saturated, leading us to conclude that the limitations to reducing the CPI value arise from the instruction-level parallelism in the workload. This aspect is discussed in Section 7.2.

All the data listed in Table 5 and depicted in the previous figures are summarized in Figure 8, wherein the graph is divided into two sections; the left-most section corresponds to the imperfect branch predictor, whereas the right-most section corresponds to perfect branch prediction. The trends in CPI discussed separately for each dimension being explored are visible jointly in this figure. Note that the configurations with out-of-order issuing of operations are always better than the class-order configurations; moreover, out-of-order with fetch/dispatch width equal to 4 outperforms even the class-order configuration with width equal to 12.

As an additional summary view, Figure 9 presents CPI adders with respect to the corresponding InfPf configurations (infinite cache and perfect branch prediction). That is, Figure 9 depicts the degradation in CPI arising from the finite parameters, for the different configurations being considered; the numbers inside the chart correspond to the adder expressed as a fraction (%) of the CPI value for configuration InfPF.

Figure 10 depicts the fraction (%) of CPI improvement (i.e., reduction in CPI) achieved by the various configurations with respect to models with fetch/dispatch width equal to 4, computed as

$$\%\ improvement = \frac{model - basemodel}{basemodel} \times 100$$

Figure 10a depicts the improvement with respect to configuration c4StBp, whereas Figure 10b depicts the improvement with respect to configuration o4StBp. This figure shows an always-increasing behavior, indicating that there is always a gain in increasing the features of the processor, without any visible saturation other than the diminishing return from increasing the processor width.

6. Utilization of resources and pipeline stalls in configuration o4StBp

In addition to the overall CPI achievable with a given processor configuration, it is important to know the utilization of its various resources and the sources of pipeline stall conditions. This information is useful to explain the trends encountered, and determine whether the resources are being utilized properly.

Our simulation environment reports periodically (every one million cycles) the number of instructions that have been fetched, as well as the number of expanded and retired operations. For example, Figure 11 illustrates the evolution of CPI in configuration o4StBp (in terms of PowerPC instructions). As inferred from this graph, the cumulative CPI evolves rather smoothly in spite of drastic changes in the instantaneous value (every 1M cycles). This type of behavior is encountered even in the case of the configurations with infinite cache and perfect branch prediction.

In addition, our simulation environment collects detailed aggregate information regarding the utilization

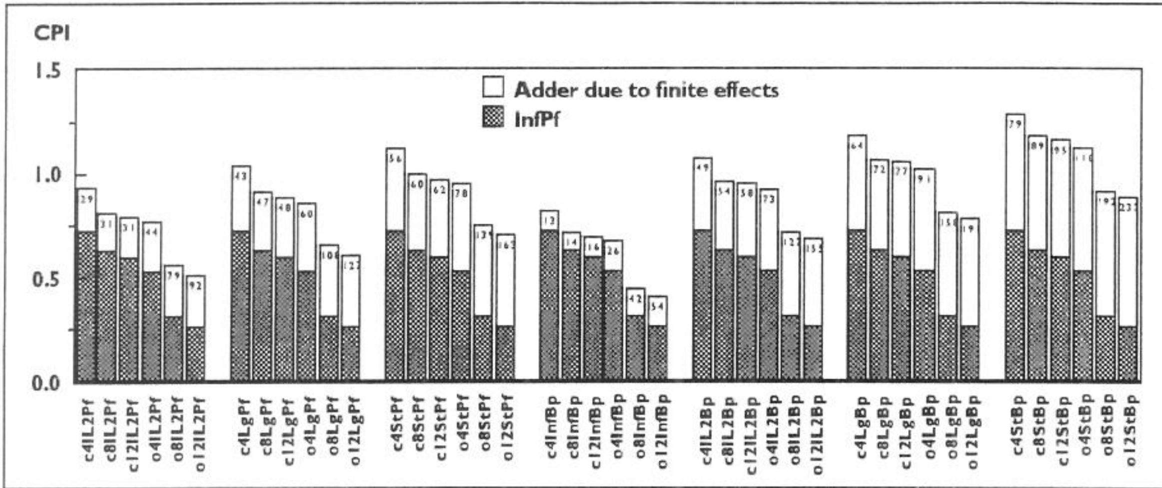
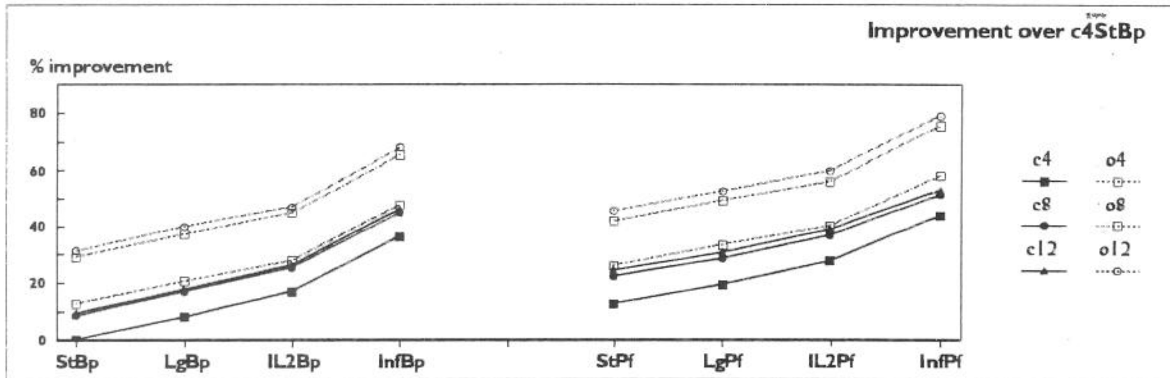


Figure 9: CPI adders due to finite effects



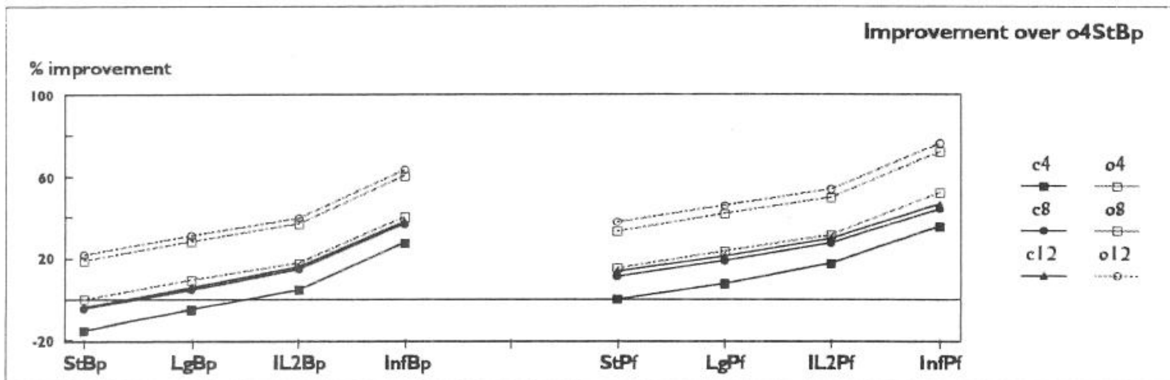


Figure 10: Improvement with respect to narrow fetch/dispatch: a) over c4StBp; b) over 04StBp

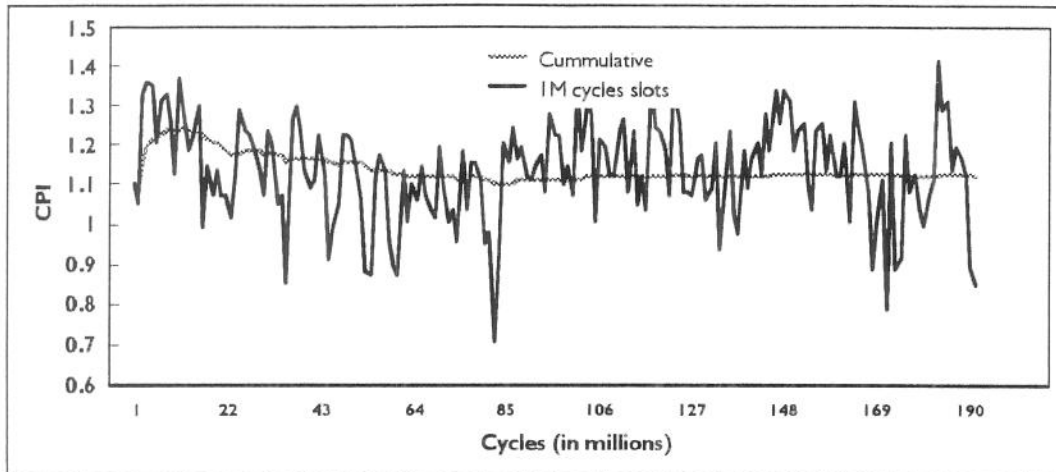


Figure 11: CPI evolution in configuration o4StBp

of various resources throughout the entire workload. We now describe this data, which are used in Section 7 to analyze the trends encountered in Section 5.

6.1 Histograms of resources' utilization

Figure 12 illustrates the utilization of the various pipeline stages, expressed as histograms of the number of cycles in which groups of instructions (fetch stage) or primitive operations (other stages) were processed simultaneously, for configuration o4StBp. As depicted in the figure, there is a large number of cycles wherein the various stages are idle (zero instructions/operations being processed). The reasons leading to the idle cycles are analyzed in more detail later. On the other hand, whenever there is activity in progress, the various stages tend to be well utilized, as indicated by the higher bars towards the right part of the graph (according to the resources of configuration o4StBp, only stages *Issue* and *Retire* may process more than four operations per cycle).

The stages' activity in configuration o4StBp is illustrated in more detail in Figure 13, which depicts the number of instructions/operations processed in each stage as a function of the size of the group in which they were handled. In other words, Figure 13 corresponds to Figure 12 but expressed as operations instead of cycles (thereby excluding cycles without activity). As shown here, most of the work is performed using large groups: most instructions are fetched in groups of four, whereas most operations are renamed in groups of four, issued in groups of three to five, and retired in groups of six.

Figure 14 illustrates the usage of the different issue queues in configuration o4StBp, which measures the number of primitive operations of each class dispatched and waiting for availability of operands and/or functional unit, where we have excluded the cycles when these queues are empty. For the workload under consideration, these queues tend to be occupied with few entries; however, the memory issue queue stays full a non-negligible number of cycles.*

Figure 15 depicts the usage of the retirement queue in configuration o4StBp, as well as a view of the distribution of primitive operations in progress (operations "in-flight"). Recall that the size of the retirement queue is 128 entries, whereas the maximum number of primitive operations in-flight is 160. This figure shows that the retirement queue has a peak at length 24 to 28, which is attributable to the expansion of instructions into multiple primitive operations. The distribution of entries in the retirement queue decreases steadily towards the queue size. On the other hand, the distribution of the number of operations in-flight has non-negligible values throughout the entire range, with higher values towards few entries in the queue.

Since the operations in-flight corresponds to the operations in the retirement queue plus the instructions which have been fetched but not yet dispatched, there are always more operations in-flight than in the retirement queue. However, this is not readily visible in the histograms in Figure 15 because of the dynamic behavior of the number of entries in the queues.

Figure 16 illustrates the usage in configuration o4StBp of the instruction buffer, the store queue and the load/store reorder buffer. This graph shows that the I-buffer tends to be occupied by few instructions (less than 7) or

* The floating-point issue queue is not depicted because the workload being evaluated has few floating-point instructions.

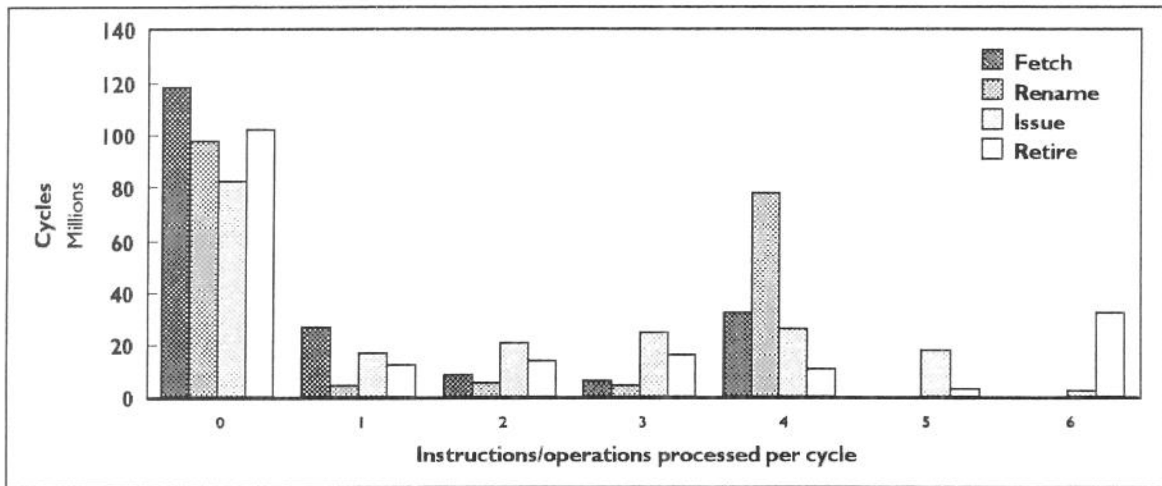


Figure 12: Utilization of pipeline stages in configuration o4StBp

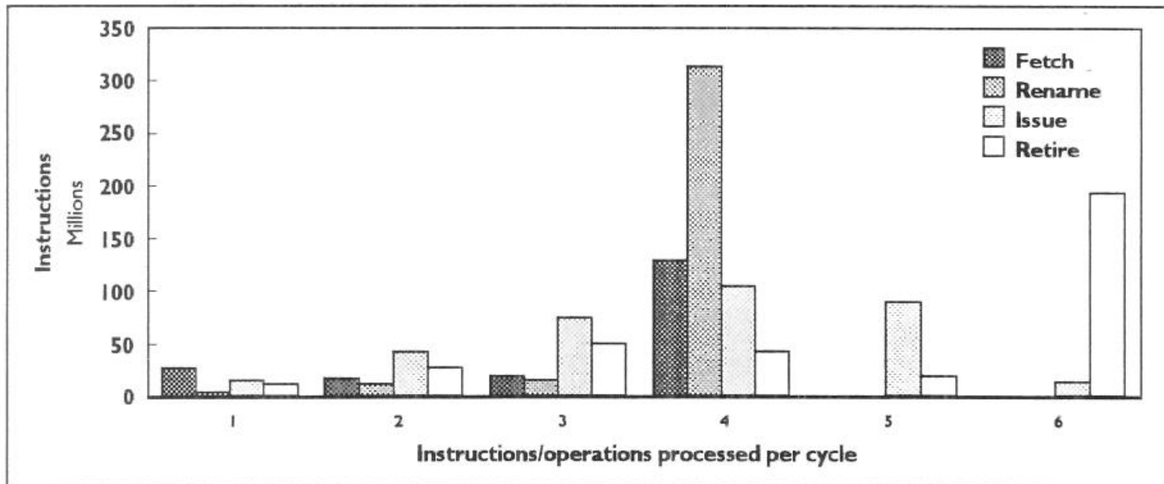


Figure 13: Width of processing operations in configuration o4StBp

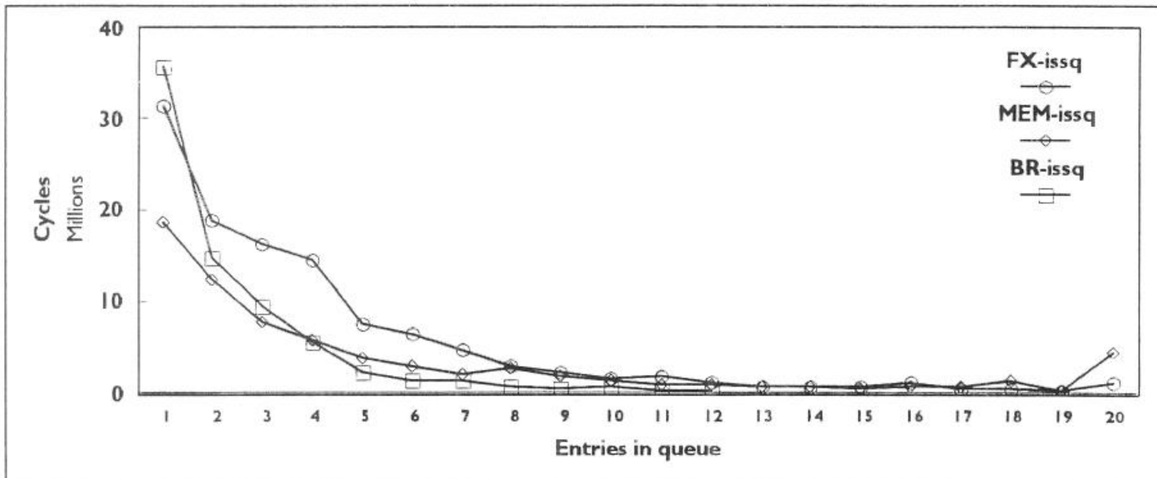


Figure 14: Utilization of issue queues in configuration o4StBp

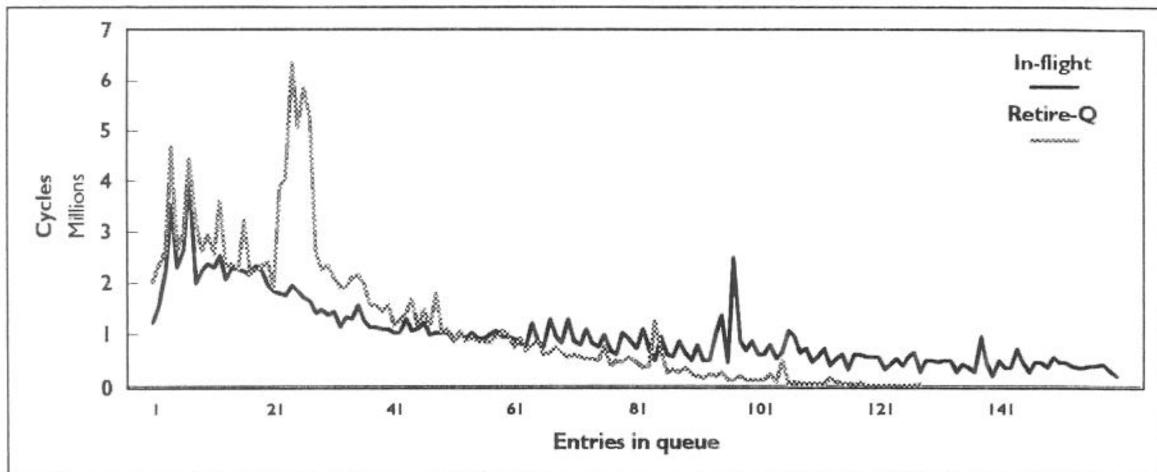


Figure 15: Retirement queue and instructions in-flight in configuration o4StBp

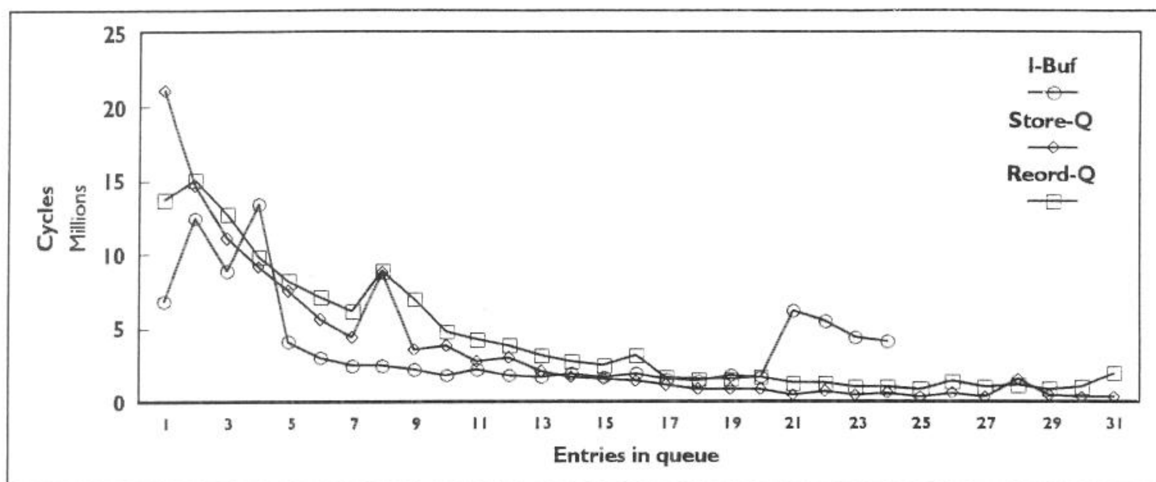


Figure 16: Usage of other queues and buffers in configuration o4StBp

by many (21 to 24). We continue investigating the reasons behind the peak in the range 21 to 24. In contrast, the store queue and the load/store reorder buffer have a decreasing trend from short to long length, although the reorder buffer has a non-negligible value at the largest entry.

6.2 Pipeline stall conditions

As depicted in Figure 12, there is a large number of cycles in which no activity takes place in the various pipeline stages. In fact, in configuration o4StBp, stages stay idle about 50% of all the cycles required to execute the instructions in the trace. A portion of this behavior is attributable to the model stalling the pipeline upon encountering a branch misprediction (whereas an actual processor would continue executing instructions along the predicted path, which would be discarded later). In addition to quantifying stalls due to misprediction, the identification of other reasons leading to idle cycles might be helpful to determine microarchitecture features that can improve performance.

We study the behavior of the processor model from the point of view of retiring instructions. That is, we investigate the reasons why it is not possible to retire the maximum number of operations in each cycle. For these purposes, the simulator keeps track of operations as they flow through the processor pipeline, and records the reason ("trauma") for an operation not making forward progress; note that, during its flow through the pipeline, an operation may experience more than one trauma.

Every cycle, the retirement logic attempts to retire the maximum number of operations possible; whenever this is not achieved, the current (most recent) "trauma" associated to the first operation that cannot be retired is recorded, tagged with the number of operations actually retired. This information is used to build histograms and *stacks* of traumas.

Note that traumas are associated with the flow of the operations through the pipeline, not the resources in the processor; consequently, a trauma assigned to an operation remains with that operation until a new trauma is encountered, regardless of the state of the processor resources. For example, an I-cache miss leads to assigning that trauma to the first operation decoded from the first instruction fetched after the miss; that trauma remains assigned unless the operation encounters another trauma.

Traumas recorded by the simulator records are grouped into 27 classes, as listed in Table 6. Traumas 1 through 9 are associated to the fetch stage, whereas traumas 10 through 12 are associated to the decode/expand and rename/dispatch stages. Traumas 13 through 16 are related to the issue stage, whereas traumas 17 through 21 are related to memory accesses. Traumas associated to dependencies among the operations are those numbered 22 through 25. Traumas 26 and 27 are related to the store queue and availability of cache ports for store operations, respectively. An additional trauma, not listed in the table, corresponds to the case of "normal trauma," which occurs whenever an operation cannot be retired but the operation has not been delayed in its progress through the pipeline by any abnormal reasons; this is a consequence of some stages, such as decode and dispatch, having lower bandwidth than the retirement stage.

Table 6: List of traumas

Id	Name	Description	Stage
1	IF_NFA	NFA misprediction	Fetch
2	IF_TLB1	I-TLB miss	
3	IF_TLB2	TLB2 instr. miss	
4	IF_L2	L2 cache instr. miss	
5	IF_L1	L1 I-cache miss	
6	IF_PREF	L1-I miss, prefetch hit	
7	IF_PRED	Branch misprediction	
8	IF_FULL	I-buffer full	
9	IF_MISC	Other reasons	
10	Decode	Cannot decode instr.	Decode
11	Rename	Cannot rename instr.	Rename/ dispatch
12	Dispatch	Cannot dispatch instr.	
13	FUL_FIX	Too many INT ready	Issue
14	FUL_FPU	Too many FP ready	
15	FUL_MEM	Too many MEM ready	
16	FUL_BR	Too many BR ready	
17	MM_OTHR	Miscellaneous reasons	Memory access
18	MM_TLB1	D-TLB miss	
19	MM_TLB2	TLB2 data miss	
20	MM_DL2	L2 cache data miss	
21	MM_DL1	L1 D-cache miss	
22	RG_FIX	Result from INT unit	Register dependencies
23	RG_FPU	Result from FP unit	
24	RG_MEM	Result from MEM unit	
25	RG_BR	Result from BR unit	
26	ST_DATA	Store data not ready	Store access
27	RET_ST	Cache port unavailable	

For example, the left-most stack in Figure 17 depicts the distribution of cycles according to the number of operations retired simultaneously, in configuration o4StBp. If six operations were retired every cycle, the average CPI would be 0.34 (1x2.01/6: the expansion factor 2.01 divided by six operations). However, as discussed earlier, this CPI value cannot be achieved because the rename/dispatch stage is limited to four primitive operations per cycle. In fact, six operations are retired only 16.7% of the cycles, as depicted by the lowermost portion of the stack.

The remaining parts of the stack, from top to bottom, depict the percentage of cycles when 0, 1, 2, etc. operations are retired simultaneously. As we have already pointed out, there is a large number of cycles in which no retiring activity takes place in this configuration, so that the topmost portion of the stack accounts for over 50% of the cycles.

The right-most stack in Figure 17 depicts an alternative view, according to the pipeline stage or unit (see Table 6) which originated the trauma assigned to the operation that determines the end of the group of operations retired simultaneously. As in the previous case, the lowermost part of the stack corresponds to the case of retiring 6 operations (the "no trauma" portion). As it can be inferred from this figure, the largest sources of traumas are the *fetch* stage, the *memory access* stage, and *dependencies* among operations.

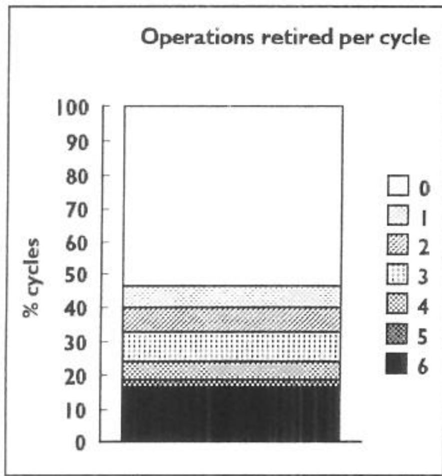
A closer examination at the sources of traumas is shown in Figure 18 for configuration o4StBp, which indicates that the most frequently encountered traumas are those associated with accesses to the second level cache (traumas IF_L2 and MM_DL2, for instructions and data respectively), followed by dependencies on load operations (RG_MM) and lack of ports for retiring store operations (RET_ST), and then dependencies on integer operations (RG_FX). The next set of traumas are accesses to the first-level data cache (MM_DL1), first-level instruction cache (trauma IF_L1), and data accesses to the second-level translation look-aside buffer (MM_TLB2). These are followed by branch misprediction (IF_PRED) and instruction accesses to the second level TLB (IF_TLB2).

The memory related traumas seen in Figure 18 are not surprising. The processor model used in this investigation includes an instruction prefetch buffer but there is no support for prefetching into L2; similarly, the model has no support for prefetching from L2 into L1-D cache, or for pre-updating the TLBs.

Additional information extractable from our simulator is a correlation among the traumas recorded and the number of unused retirement slots due to those traumas. That is, the trauma associated to the first operation that could not be retired in a cycle can be weighted by the number of empty retirement slots for that cycle. For example, Figure 19 depicts the frequency of traumas shown in Figure 18 (for configuration o4StBp), weighted by the number of unused retirement slots. Note that the largest sources of traumas are the same as the ones in Figure 18, but the ratio between the bars is larger. This illustrates, for instance, that an L2 miss trauma is more expensive than others (it leads to cycles without any instruction being retired).

The data in Figure 19 corresponds to an approximation (worst case condition) to the actual retirement opportunity lost. In practice, not all the unused retirement slots in a given cycle are really a loss, because some operations after a trauma may themselves be subject to a trauma, so they could not have been retired anyway.*

* We are currently modifying our simulator to provide the actual number of operations ready to retire which follow the first operation that cannot be retired in a given cycle.



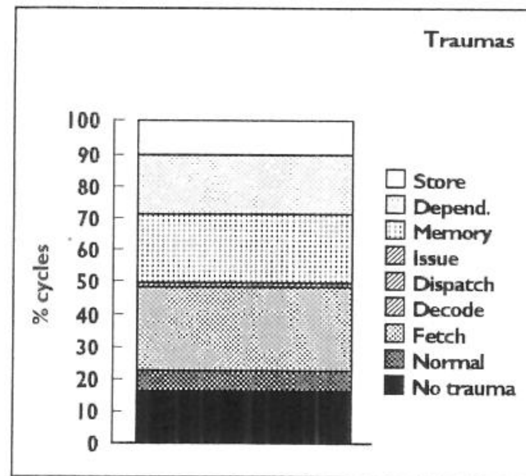


Figure 17: Normalized trauma stacks in configuration o4StBp:
a) number of operations retired per cycle; b) sources of traumas

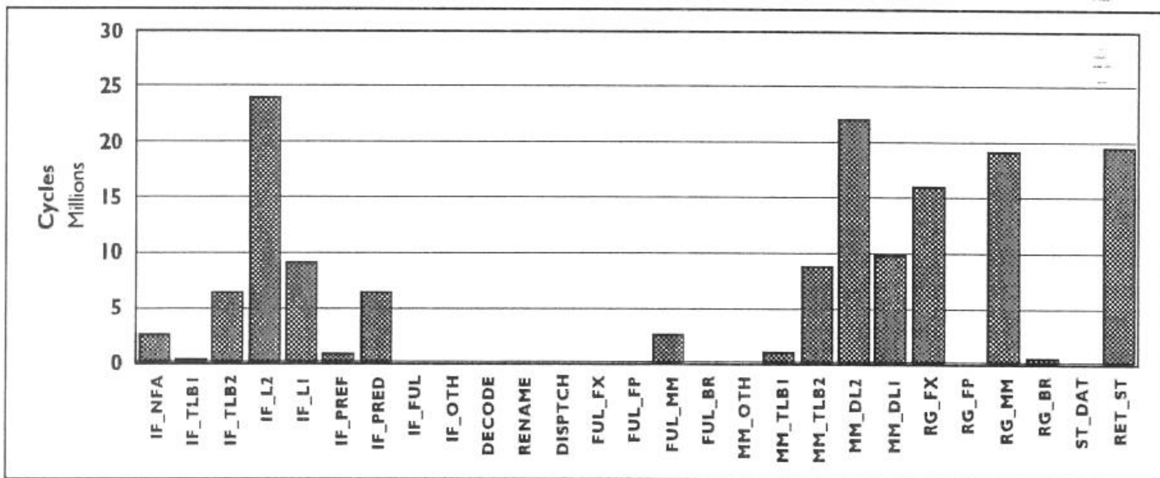


Figure 18: Histogram of traumas in configuration o4StBp

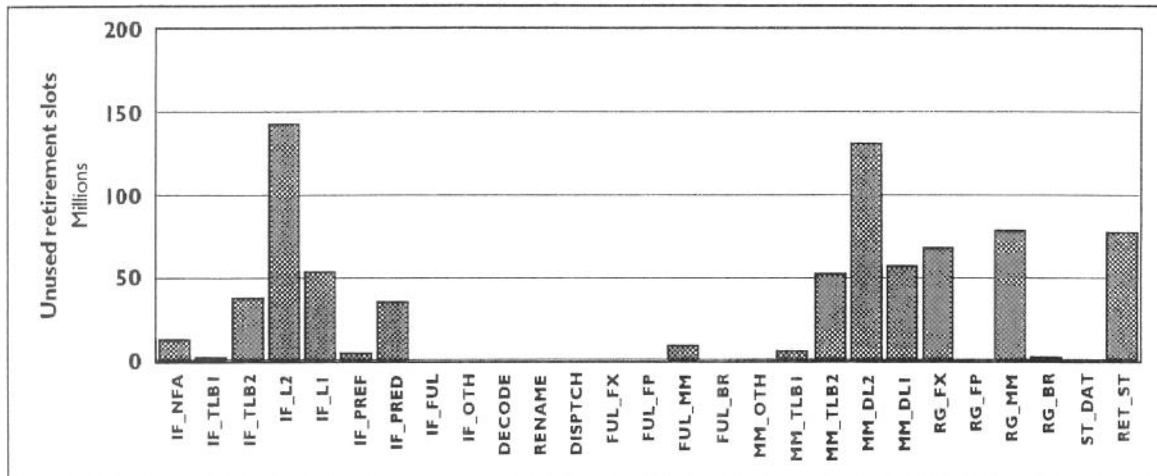


Figure 19: Histogram of traumas weighted by loss of retirement opportunity in configuration o4StBp

7. Analysis of performance trends

We now use the traumas' data collected from our simulation environment to analyze and explain the performance trends encountered in the configurations explored.

7.1 Effects of second level instruction cache in configuration o4StBp

Let us first consider the effects of the second level cache, the most frequent trauma in configuration o4StBp (see Figure 18). For these purposes, let us explore the traumas for configuration o4IL2Bp, which is the same as o4StBp but excluding effects from L2 cache misses (infinite size L2). The difference in CPI value for these two configurations (1.12 and 0.93) is due to 33.8 million more cycles in the case of o4StBp; however, traumas due to the second level cache account for 46 million cycles in Figure 18 (approximately 24 million from IF_L2 and 22 million from MM_DL2).

Figure 20 depicts the histograms of traumas for configurations o4StBp and o4IL2Bp, including an entry for the number of cycles when there was no trauma (i.e., the maximum possible number of operations was retired; this is the last set of bars in the graph, with the label NONE). As expected, the traumas associated to the L2 cache have disappeared in configuration o4IL2Bp, while the relative importance of the other traumas remains almost the same as in the case with finite L2 cache. The frequency of several traumas has increased (e.g., IF_L1, IF_TLB2, MM_DL1, MM_TLB2), which indicates that the degrading effects of some L2 traumas have shifted to other traumas. In other words, some L2 traumas either are or precede operations that are affected by other traumas, so that eliminating the L2 trauma makes visible the others; this explains the difference between the 33.8 and 46 million cycles mentioned above.

The processor model used in this study does not have parameters to change the associativity of the cache organization, so from these experiments we cannot conclude whether the degradation arises from capacity or compulsory misses in the L2 cache. In any case, we can postulate that any improvements in the size or associativity of this subsystem leads to a reduction in its frequency of traumas, thereby affecting the final performance.

7.2 Diminishing improvement in wider configurations

Let us now address the diminishing performance improvement obtained when increasing the decode/dispatch width to twelve operations per cycle. This phenomena was illustrated in Figure 7.

Figure 21 depicts the histograms of traumas for configurations o4StBp, o8StBp, and o12StBp. This figure shows that the *frequency of traumas is basically the same for the three configurations,* with the only exception of trauma RET_ST which corresponds to lack of ports to retire store operations. In addition, the number of cycles when there was no trauma differs drastically among the three cases; this can be expected because the probability of finding as many operations ready to retire as the retirement width is lower for wider configurations.

In terms of the number of cycles required to process the workload, there is a difference of approximately 35 million cycles between configurations o4StBp and o8StBp, but only 5 million cycles between configurations

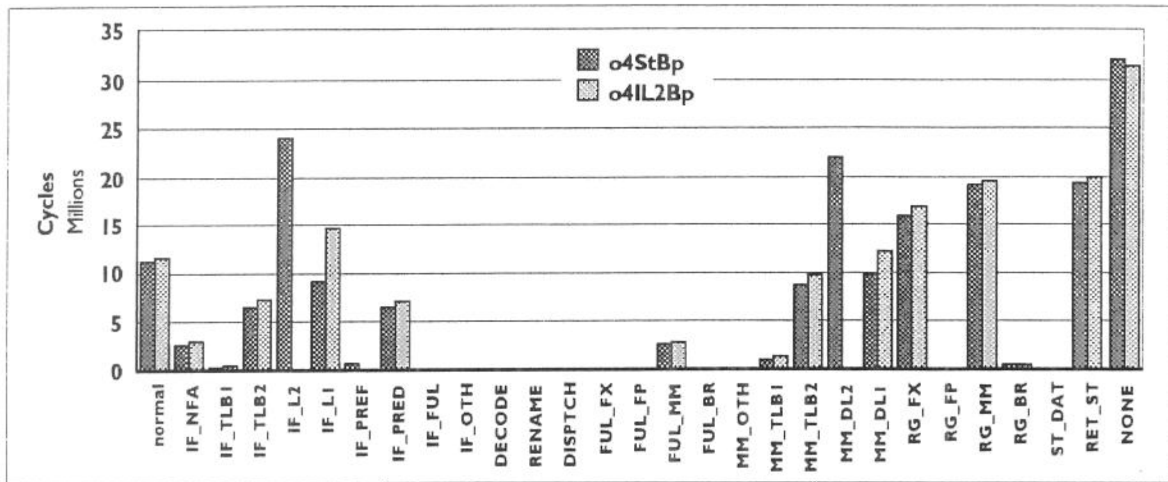


Figure 20: Histograms of traumas in configurations o4StBp and o4IL2Bp

o8StBp and o12StBp; these differences account for the corresponding CPI improvements (see Table 5). From inspecting Figure 21, it follows that trauma RET_ST accounts for about 10 million more cycles in o4StBp than in o8StBp, whereas there are about 20 million more cycles associated to the no trauma condition. In contrast, each of these factors contributes about 5 million cycles between configurations o8StBp and o12StBp.

The traumas data indicate that there is a bottleneck in the data cache ports. Since configuration o4StBp has only two memory units, there are only two cache ports. Given the frequency of load and store instructions in the workload, and the higher priority assigned to load over store operations, store operations are required to wait for the availability of cache ports.

On the other hand, configuration o8StBp has twice as many memory units and therefore cache ports, thereby alleviating the bottleneck. Although further increases in the number of memory units and ports continues decreasing the frequency of trauma RET_ST as well as the frequency of no traumas, these reductions are counterbalanced by small increases on other factors (such as traumas Normal, IF_PRED, RG_FX, RG_MM).

As further exploration of this topic, Figure 22 depicts the histograms of traumas for the same configurations as above but assuming that each one has twice as many cache ports (these configurations are not included in Table 5). That is, each configuration uses separate ports for store operations, and there are as many extra ports as the number of memory units. As it could be expected, the effects of trauma RET_ST have almost disappeared; however, the CPI achieved in each case is only about 1% different from the earlier case. In other words, the effects of trauma RET_ST have been shifted to other traumas; in particular, RG_FX and RG_MM have increased in all configurations, whereas FUL_MM has increased in configuration o4StBp.

These results indicate that operations affected by trauma RET_ST closely precede operations that are affected by register dependencies, so that retiring the store operation makes visible the trauma in the operations that follow but does not reduce the number of cycles required by the workload. The new traumas now visible are mostly dependencies among operations, which indicates that there is a substantial increase in extraction of instruction-level parallelism (ILP) when moving from configuration o4StBp to configuration o8StBp, but further substantial gains in ILP are not possible.

7.3 Effects of class-order issuing of operations

Let us analyze now the effects of class-order issuing of operations, by comparing the traumas in this case with those arising in the case of out-of-order issue. The performance degradation arising from class-order issuing was depicted in Figure 4.

Figure 23 depicts histograms of traumas for configurations o4StBp and c4StBp, that is, for configurations with identical resources but just different issue policy. As illustrated in the figure, the frequency of most traumas is very similar, with the dramatic exception of RG_FX; in the case of class-order issuing, trauma RG_FX is almost four times more frequent that in the case of out-of-order issuing. This indicates that many integer operations in the program are independent but appear in sequential program order, so that they are not issued in the class-

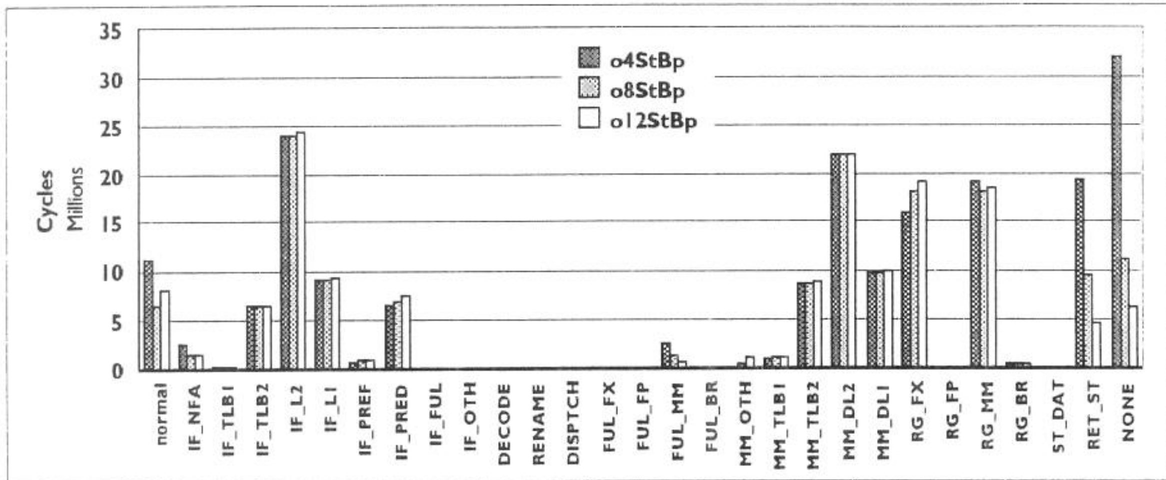


Figure 21: Histogram of traumas for configurations o4StBp, o8StBp, and o12StBp

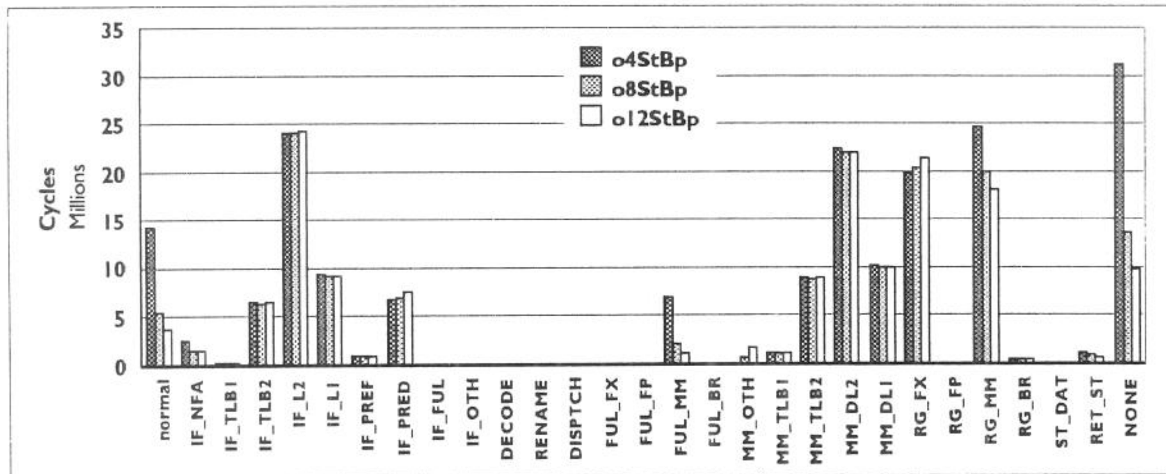


Figure 22: Histogram of traumas for configurations o4StBp, o8StBp, and o12StBp
with twice as many data cache ports

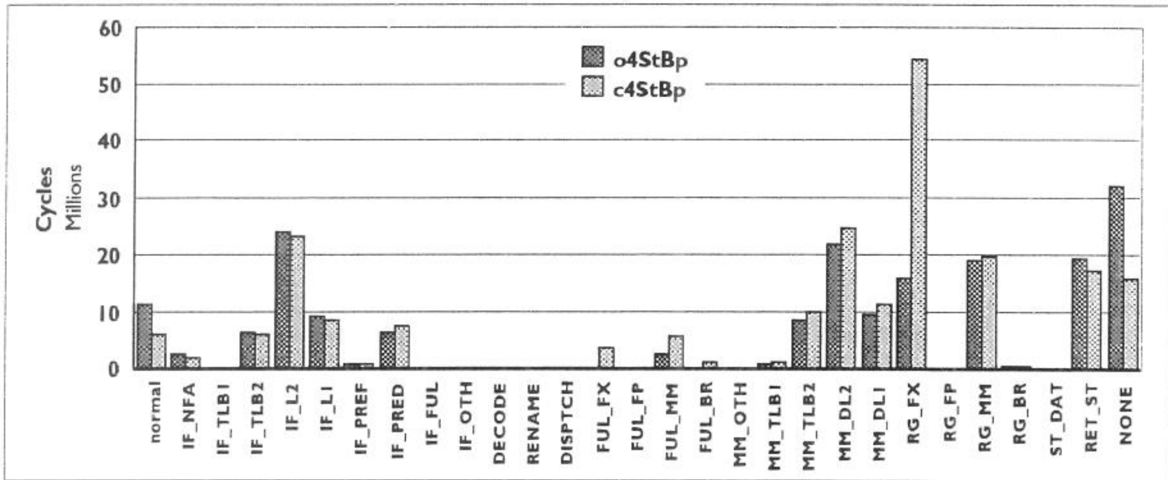


Figure 23: Histogram of traumas for configurations o4StBp and c4StBp

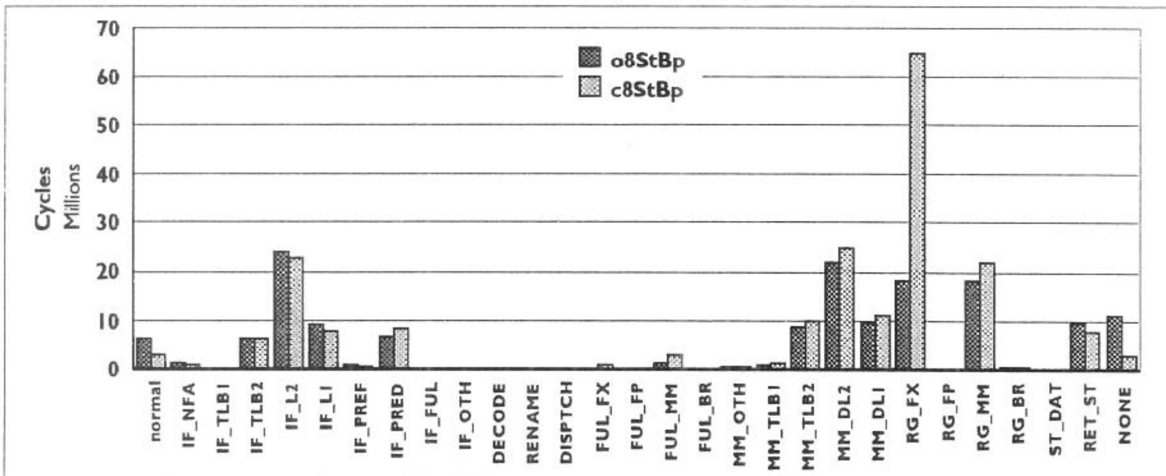


Figure 24: Histogram of traumas for configurations o8StBp and c8StBp

order policy but they are dynamically reordered in the out-of-order policy. Moreover, the RG_FX traumas in configuration c4StBp do not mask other conflicts, because their removal does not lead to a substantial increase of other traumas. A similar behavior is found among configurations o8StBp and c8StBp, as depicted in Figure 24.

8. Effects of microarchitecture features

We now explore the effects of some microarchitecture features. In particular, we explore the effects of the following features on configurations o4StBp and o8StBp:

- lack of next-fetch-address prediction;
- lack of early branch resolution;
- double the number of instructions fetched per cycle;
- one fewer pipeline stage for load operations;
- one or two additional decode stages in the pipeline;
- four times larger TLBs; and
- twice larger caches.

Table 7 lists the CPI values as well as the relative change with respect to the original value for each configuration. Removing features such as next fetch address prediction and early branch resolution account for 3.6 to 6.6 % CPI degradation, whereas doubling the fetch width or removing one pipeline stage for load operations accounts for about 1 to 2 % improvement. Similarly, one additional decode stage in the pipeline leads to about 2 % degradation, whereas two extra decode stages imply 2.7 to 4.4 % degradation. On the other hand, making the TLBs four times larger (i.e., 512, 4096 entries, respectively) improves performance by 3.6 to 4.4%, whereas a cache twice as large provides 8.9 to 11 % improvement.

The data listed in Table 7 indicate that, individually, most of these features do not provide a dramatic advantage in performance; instead, each feature helps by a small factor, making necessary to incorporate several of them for a more substantial performance gain. Among the features explored, larger caches and early branch resolution are the most effective ones, whereas one additional decode stage might not be too detrimental.

Table 7: Effects of some microarchitecture features

Feature	o4StB		o8StBp	
	CPI	% change	CPI	% change
Original	1.12	-	0.91	-
No NFA predictor	1.16	-3.6	0.95	-4.4
No early branch resolution	1.18	-5.4	0.97	-6.6
Double I-fetch bandwidth	1.10	1.8	0.90	1.1
One fewer cycle in load operations	1.11	0.9	0.89	2.2
One additional decode stage	1.14	-1.8	0.93	-2.2
Two additional decode stages	1.15	-2.7	0.95	-4.4
Larger TLBs (4x)	1.08	3.6	0.87	4.4
Larger caches (2x)	1.02	8.9	0.81	11.0

9. Concluding remarks

We have investigated the potential performance of PowerPC-based wide superscalar processors on a standard OLTP workload, using a PowerPC 601 instruction and data reference trace containing 172 million instructions. We have explored instruction-level parallelism as a function of the policy for issuing instructions, the processor width, the size of the cache memory, and the branch predictor. We have studied factors that limit increasing performance in wide processors (i.e., factors limiting the reduction of the average number of cycles required per instruction, CPI), from the perspective of retiring operations. The analysis has been carried out through the use of stacks and histograms indicating the performance degradation (with respect to more ideal cases) contributed by different sources. We have also explored the sensitivity of two configurations to selected microarchitecture features.

The simulation results have shown that, on the workload and processor organizations considered, issuing operations in class-order in narrower configurations with finite caches and branch predictor leads to a 15 to 30% degradation with respect to issuing them out-of-order. This degradation can increase up to 125%, for the case of wider processors with infinite caches and perfect branch prediction. In other words, the degradation arising from the class-order issue policy becomes more severe as other features of the processor improve. Moreover, since class-order is less restrictive than in-order issuing, the degradation can be expected to be even more severe when using a fully in-order policy, though our environment does not have the ability to quantify such degradation exactly.

The simulation results also show that a branch prediction table with 8192 2-bit counters degrades performance with respect to a perfect predictor in the range from 18 to 26% in smaller out-of-order implementations; such a degradation may increase up to 54% for wider configurations with infinite caches. This suggests that branch prediction improvements are more useful for improving performance in the case of wider out-of-order configurations.

Moreover, the results indicate that a processor dispatch width of eight operations per cycle is advantageous, whereas wider organizations provide diminishing performance improvement. This saturating effect arises from the availability of instruction-level parallelism in the workload, not from limitations in the processor pipeline. In fact, the results indicate that pipeline resources do not get overloaded or saturated.

Consequently, as inferred from Table 5, doubling the features of a processor whose dispatch width is four by doubling the number of units and the various widths, while preserving the size of caches and prediction accuracy, produces approximately 20% overall performance improvement. Doubling also the size of the caches produces an additional 10% improvement. Further enhancements in cache size and branch prediction accuracy can increase these factors even further. The overall possible gain from the wider implementation, infinite caches and perfect branch prediction can decrease CPI from 1.12 down to 0.31.

Data collected from the simulations indicate that, in general, the activity in the processor on the workload considered tends to be in bursts. There are many cycles of almost total inactivity in the various stages, but there also are many cycles in which the stages are fully busy.

As it could be expected, one of the bottlenecks to higher performance is found in the memory subsystem, in particular in delays when misses are encountered in the cache and the translation look-aside buffer, thereby validating common wisdom regarding the effects of memory on OLTP workloads.

The exploration of performance sensitivity to selected microarchitecture features shows that the various features contribute to overall performance, but the improvement associated with each of the features is rather small (in the range from 1 to 5%). The most sensitive features are the size of the cache memory, as already demonstrated by the other data, and the early resolution of branches (at the dispatch stage).

The experimentation described in this report has been carried-out using a set of tools developed with the objective of supporting exploration of microarchitecture features, by providing the ability to modify a reasonably large set of parameters. In addition, these tools allow simulating in excess of 100 million processor cycles per hour. The combination of these two features, parameterization and speed, has enabled the extensive exploration of microarchitecture features. The data reported here are just a small sample of the entire exploration space made possible by the tools.

Acknowledgments

We thank Eric Kronstadt, Al Chang, Dan Prener, and Dave Meltzer, for their contributions and support to this work. In addition, we thank Charles Moore for his suggestions to analyze performance from the perspective of retiring instructions and the generation of the traumas-based histograms and stacks, and Mary Mosher for providing the trace used in the experiments.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[2] Transaction Processing Performance Council (TPC), *TPC Benchmark C, Standard Specification*, 1993.

[3] *Fast simulation of computer architectures*, T. Conte and C. Gimarc, Eds., Kluwer Academic Publishers, Boston, MA, 1995.

[4] P. Bose, *Performance analysis and verification of superscalar processors*, Research Report RC-20094, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1995.

[5] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver, Colorado, pp.206-218, June 1997.

[6] M. Franklin, W. Alexander, R. Jauhari, A.M.G. Maynard, B. Olszewski, "Commercial workload performance in the IBM POWER2 RISC System/6000 Processor," *IBM Journal of Research and Development*, Vol. 38, No. 5, pp. 555-561, September 1994.

[7] R.J. Eickemeyer, R.E. Johnson, S.R. Kunkel, M. Squillante, S. Liu, "Evaluation of multithreaded uniprocessors for commercial application environments," *Proceedings of the 23th Annual International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, pp.203-212, May 1996.

[8] Standard Performance Evaluation Corporation, *SPEC95 Benchmark Suite*, August 1995.

[9] M. Mosher, "Validation of a PowerPC 601 TPC-C instruction and data reference trace," IBM Austin, March 1996 (internal report).

[10] P. Emma, "Understanding some simple performance limits," to be published in *IBM Journal of Research and Development*, May/June 1997.

[11] J. Moreno, M. Moudgill, J.D. Wellman, P. Bose, L. Trevillyan, "Performance exploration of PowerPC-based wide superscalar processors," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[12] J. Moreno, M. Moudgill, J.D. Wellman, "The MET: a microarchitecture exploration toolset," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[13] IBM Corporation, *RS/6000 43P Model 140,* Order No. G2217015, 1996.

[14] M. Moudgill, J. Moreno, "Turandot: a wide-issue superscalar processor model for microarchitecture exploration," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[15] J. E. Smith, "A study of branch prediction strategies," *Proceedings of the 8th International Symposium on Computer Architecture,* pp. 443-458, May 1981.

[16] J.D. Wellman, "Aria: a micro-tracing engine," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[17] P. Bose, L. Trevillyan, "Validation of a wide-issue superscalar processor model," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

Copies may be requested from:

IBM Thomas J. Watson Research Center
Publications Office, 16-220
Post Office Box 218
Yorktown Heights, NY 10598

Some reports are available via the
Cyberjournal on the WWW.
http://www.watson.ibm.com:8080