RC22454 (W0205-159) May 20, 2002
Computer Science

IBM Research Report

Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor

F. N. Eskesen, M. Hack, T. Kimbrel, M. S. Squillante
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

R. J. Eickemeyer, S. R. Kunkel
IBM Server Group
Rochester, MN 55901

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor

F.N. Eskesen,[†] M. Hack,[†] T. Kimbrel,[†] M.S. Squillante,[†] R.J. Eickemeyer,[‡] S.R. Kunkel[‡]
[†] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
[‡] IBM Server Group, Rochester, MN 55901

Abstract

Simultaneous multithreading (SMT) is an approach to address the well-known problems of memory accesses increasingly dominating processor execution time and of limited instruction level parallelism. Previous research has explored the benefits and limitations of SMT based on specific processor architectures under a variety of workloads. In this paper, we present a performance analysis of SMT in a PowerPC-based wide superscalar processor architecture under a broad range of workloads, which includes combinations of TPC-C, SPECint and SPECfp. Although some of our results are consistent with previous work, our results also demonstrate some differences and we use these results to explore and identify the primary causes of such differences. This includes an investigation of thread characteristics that work well together in SMT environments, and thread characteristics that do not work well together.

1 Introduction

The design of current and future processor architectures depends upon a large number of key tradeoffs and technology issues. Two such tradeoffs/issues, which continue to be the focus of current and future processor designs, are due to memory accesses increasingly dominating the execution time of processors and to the limited degree of instruction-level parallelism. One approach to addressing these well-known problems is called simultaneous multithreading (SMT), in which multiple threads are allowed to issue multiple instructions every cycle.

A number of studies in the research literature have explored various aspects of SMT. This includes an evaluation of the benefits and limitations of SMT within the context of a specific processor architecture, such as fairly recent high-performance, out-of-order, superscalar processor architectures [13, 5, 12, 2]. A broad range of application and system workloads have been considered in many of these studies, including workloads based on database management systems [7]. SMT has also been considered for other uses, such as the speculative execution of multiple paths of execution [15] and the recovery of transient faults [11, 14]. Furthermore, the Intel Xeon processor MP family features a version of SMT [6].

In contrast, no previous research has investigated the benefits and limitations of SMT within the context of a PowerPC-based wide superscalar processor architecture. One of the primary objectives of this paper, therefore, is to present such a performance analysis of SMT under a broad range of workloads, including various combinations of TPC-C, SPECint and SPECfp. There are several important differences between the PowerPC-based architecture of interest in our study and the processor architectures of previous studies and we explore the impact of some of these differences on SMT design and performance issues. These issues are examined through detailed trace-driven simulation. We have made significant modifications to the Turandot simulator [9, 10] , which models a superscalar processor based on the PowerPC architecture, in order to support SMT. Following the operatic theme of our colleagues, our SMT version of the Turandot simulator is dubbed *Figaro* reflecting the competition between Figaro and the Conte di Almaviva for the affection of Figaro's fiancee Susanna.

The present study was originally started as an immediate follow-on to some of our previous research in the area of coarse-grained multithreading [4, 3]. Coarse-grained multithreading was the natural choice in this previous work because the processor of interest executed instructions in order [1]. However, our follow-on study turned to consider an existing out-of-order PowerPC-based processor model (i.e., the one supported by Turandot), and thus SMT is the natural choice. We further sought to explore the benefits and limitations of SMT within the context of such an existing wide superscalar architecture that is not optimized in any major way to maximize SMT performance. Given the focus of this workshop on duplicating and deconstructing previous results, we primarily consider instances of this PowerPC-based processor model parameterized under technology assumptions that are consistent with those of previous work in the area. Our on-going research also considers instances of this suboptimal SMT processor model under assumptions based on current and next-generation technology.

Although some of the results from our study are consistent with previous work, our results also demonstrate some differences. In particular, while our results illustrate similar performance trends to previous work, the magnitude of performance improvements is generally smaller and in certain instances we observe performance degradation. Our analysis explores some of the primary causes of these differences. This includes an investigation of thread characteristics that work well together in SMT environments (such as those found in SPECint and TPC-C), thus providing significant performance benefits, and thread characteristics that do not work well together (such as those found in SPECfp), thus providing smaller improvements and in some cases resulting in performance degradation. To quantify these effects, we introduce the notions of application citizenship and selfishness. Our results also demonstrate and quantify the performance impact of different design parameters on the benefits and limitations of SMT within the context of PowerPC-based processors. Our results illustrate some of the complex tradeoffs that must be considered to extend an existing processor design in order to incorporate SMT, as well as to design an SMT processor from the ground up. It is our hope that some of the lessons learned in this study will be useful when attempting to optimize a processor design for SMT.

Unlike most previous studies, we focus on the more modest and near-term realizable case of a processor with two hardware threads. This is consistent with the number of threads used in a few recent processors that exploit multithreading [1, 6]. Furthermore, this choice allows us to carry out a full spectrum of experiments on a relatively large set of benchmarks, pairing each benchmark with each of the others. We are then able to make interesting observations about the behavior of particular combinations of benchmarks by analyzing the trends thus obtained.

The remainder of this paper is organized as follows. Section 2 presents the processor architecture and workloads used in our study, as well as the Figaro simulator. A representative set of our results are then presented, followed by some concluding remarks in Section 4.

2 Processor Architecture and Workloads

We briefly summarize in turn the processor architecture and trace-based workloads considered in our study. Then we describe our modifications and enhancements to the Turandot simulator in order to evaluate SMT performance.

2.1 Processor architecture model

Turandot [9, 10] simulates a PowerPC-based superscalar processor architecture with 32 general-purpose registers (GPRs), 32 floating point registers (FPRs), 8 condition code registers (CCRs), and 32 special-purpose registers (SPRs). The number of physical registers is larger than the number of architected registers for register renaming. In Figaro we simulate 128 GPRs, 128 FPRs, 128 SPRs and 64 CCRs, each of which has separate multiported register files.

Our processor organization has 2 fixed-point units, 2 floating-point units, 2 memory load/store units, and 1 branch unit. Each unit has a separate path to place its result in the corresponding destination. The maximum number of instructions fetched in a cycle is 8, the maximum number dispatched is 9, and the maximum number retired is 6. Decode requires three cycles, and the width of this portion of the pipeline is 4. Note that the PowerPC instruction set architecture includes certain complex instructions, such as load/store-multiple, load-with-update and string instructions, which are decomposed in the decode/expand stage into multiple simpler operations, or micro-operations. The L1D miss queue size is 8. The retirement queue has a maximum size of 100, and the maximum number of inflight instructions is 150. The pipeline has a single fetch stage, 3 decode/expand stages, a rename/dispatch stage, an issue stage, a register read stage, an execute stage, a write back stage, and a retire stage. The floating point unit adds 3 cycles to this, and a load or store that hits in the data cache adds 2. The branch prediction table consists of 16K single-bit histories.

The memory system has the following characteristics:

ITLB/DTLB: separate 128-entry, 2-way set-associative first level instruction and data translation look-aside buffers;

TLB2: a unified, 1024-entry, 4-way set-associative second level translation look-aside buffer;

L1I: a 2-way set-associative 64-KByte first level instruction cache with line size 128 bytes;

L1D: a 2-way set-associative 32-KByte first level data cache with line size 128 bytes;

L2: a unified, 4-way set-associative 2-MByte second level cache with line size 128 bytes.

The set of miss latencies in these cache structures are provided in Table 1. The base configuration parameters are for the most part unchanged from the original Turandot processor model. Moreover, they are consistent with those used across the wide range of previous SMT studies. The number of renaming registers is slightly larger than the original Turandot model (128 vs. 96), and the I and D caches were made 2-way associative. The instruction buffer is shared between the two threads, so that a stalling thread can fill up the buffer and hinder the progress of the other thread. Thus, the architecture is not optimized for SMT, and this explains in part our somewhat pessimistic results.

ITLB	DTLB	TLB2	L1I	L1D	L2
10	7	40	11	10	80

Table 1. Set of Cache and TLB latencies

2.2 Workload

We use PowerPC 601 instruction and data reference traces of the TPC-C benchmark running under AIX together with DB2, seven SPEC95 integer benchmarks (compress, gcc, go, ijpeg, li, m88ksim, perl), and five SPEC95 floating point benchmarks (hydro2d, mgrid, su2cor, swim, tomcatv), compiled by the xlc compiler. These traces were collected by the Server Group Performance Team at IBM Austin. A subset of the SPECfp benchmarks is chosen in order to keep the number of combinations of pairwise runs to a reasonable number. We exclude the SPECint program Vortex from our pairwise comparisons because we found that in SMT mode, its TLB performance degraded terribly due to orders of magnitude more misses. This appears to be an artifact of the compiler's data layout, and rectifying the problem is beyond the scope of this study. However, we do explore the causes of these problems in isolation (see Section 3.1).

Our traces are of two different kinds:

1. Each SPEC trace is for a single process, with no kernel code, and no segment escapes – each segment is assumed to be a unique entity. (The PowerPC ISA includes address segmentation. At any time there are 16 active segments indexed by a set of registers, and each instruction or data address references one of these. A segment escape changes the value of a segment register.) Each trace is stitched together from sampled chunks of two million instructions each, in order to capture the behavior of the entire run of the benchmark rather than of only a single phase.

2. The TPC-C trace is for multiple processes, includes kernel code, and uses segment escapes extensively. It also consists of stitched-together two-million-instruction chunks, but the gaps between chunks are as small as the trace tool could manage, so that the trace represents a nearly-continuous stream of execution.

2.3 Modifications to Turandot simulator

The Turandot simulator was written by Mayan Moudgill [9, 10] in order to experiment with different machine configuration parameters. Turandot simulates cycle-by-cycle, keeping track of instructions as they progress through the instruction pipeline and of the associated state of TLBs and data caches. Instructions are pre-decoded in order to increase the speed of the simulation. Data values are never simulated (and in trace-driven mode they are neither available nor necessary).

We modified Turandot primarily by supporting multiple threads to be used as instruction sources. Here the term *thread* is used in the hardware context, representing an instruction source and associated register set.

2.3.1 Basic modifications

In addition to the obvious changes made to support multiple threads (which we omit in the interest of space), the following modification to the base Turandot simulator was necessary to support SMT. We observed via an analysis of our traces that the PowerPC `isync` instruction was more prevalent in the TPC-C trace than originally expected. Our initial implementation of Figaro would drain the entire queue (for all threads) when an isync was encountered. We were able to boost SMT performance significantly (4.8%-6.3%, depending on the latency set) by allowing other threads to continue fetching if one thread is blocked with an isync. We allow alternate threads to fetch when one thread is processing the isync instruction.

2.3.2 Branch misprediction

Our simulator models the effects of branch misprediction more accurately than the base Turandot simulator. We found this to be necessary to avoid overly optimistic projections for SMT, since wrong-path instructions from a mispredicting thread interfere with those of the other thread(s). We integrated the Intermezzo trace augmenting tool [10] into our simulator to simulate wrong-path instructions.

Extending *Intermezzo* for multithreading required a plausible mechanism for purging mispredicted instructions from the hardware pipeline after a mispredicted branch instruction was executed. We chose to pessimistically keep purged instructions in the pipeline, consuming resources, until retirement.

2.3.3 Thread selection

We call any thread that is not stalled on an I-cache miss *ready*. On a given cycle, instructions are fetched from a single ready thread in the instruction fetch stage. We have experimented with the following thread selection policies.

Next-ready. This policy alternates between threads that are ready. If a reference misses in the cache but hits in the prefetch buffer (which requires one extra cycle), the thread is allowed to hold on to the fetch unit and complete its fetch on the next cycle.

Priority. One thread is given priority for a fixed number of cycles. Other threads are given control only when the priority thread cannot fetch. After the fixed number of cycles, the next thread gets a turn as the priority thread.

Every-cycle. The next thread is selected for fetching whether or not it is ready.

Coarse-grain. The current thread continues to be selected until it encounters a miss in both the instruction cache and the prefetch buffer, at which point selection starts from the next ready thread. When both threads are waiting on a miss, selection resumes with the first thread to have its outstanding miss satisfied.

Instruction count. Instructions are fetched from the thread that currently has the fewest instructions in the pipeline. This policy attempts to improve throughput by favoring threads whose instructions move quickly through the pipeline. This policy gave the best performance in a previous study [12].

Operation count. Similar to the previous policy, but counting expanded micro-operations rather than instructions.

Branch count. The thread with the fewest unresolved predicted branches is selected. This policy attempts to maximize the number of useful instructions fetched by avoiding those which are likely to be wrong-path instructions.

Miss count. The thread with the fewest outstanding cache misses is selected. This policy is designed to favor threads which are performing well in the cache and thus likely to maximize pipeline utilization.

A few subtle issues arise with the thread selection policies which depend upon where thread selection decisions are made in the simulator. In particular, if the SMT fetch unit under the next-ready policy alternates between ready threads every cycle without considering effects further down in the pipeline, then the fetch unit switches between ready threads regardless of whether any instructions were fetched on the previous cycle. Thus, if the instruction buffer fills and causes the fetch unit to stall for an odd number of cycles, successive fetches will come from the same thread even though the other thread is ready; the fetch unit was not ready when it was the second thread's turn, and so it "loses out." This may cause one thread to be starved, and therefore we refer to this policy characteristic as "unfair". On the other hand, a policy that switches threads only when instructions are actually fetched in the previous cycle is called "fair". Interestingly, under the TPC-C workload Figaro shows that the unfair next-ready selection policy outperforms the fair next-ready policy slightly. The unfair policy results in higher utilization of the fetch buffer, and there are more stalls due to the maximum number of inflight instructions being reached than under the fair policy. The unfair fetch unit does a better job of keeping the pipeline full. A natural additional hypothesis for the improved performance was that the unfair policy, by staying with one thread longer, exhibits better locality. This hypothesis, however, was not borne out by the cache and TLB miss statistics.

To examine these issues in more detail, Figaro divides the selection mechanisms into their basic components. Specifically, the "fair" or "unfair" selection decisions are considered only if the I-fetch unit is ready. The fair policy does not allow a thread switch in a cycle where no instructions were fetched (for whatever reason) but the thread appears to be ready, whereas the unfair policy will allow this switch. Fair and unfair apply to all selection mechanisms.

When instructions are fetched, the fair and unfair policies are identical. When no thread switch occurs, there is also no policy difference. Thus, it is only necessary to focus on cycles where no instructions were fetched and a thread switch occurred. While fetching may not occur for several reasons, our analysis of the trace-based workloads shows that such cases occur relatively infrequently. Hence, the differences between fair and unfair are relatively small, though nonnegligible in some cases.

2.3.4 Page mapping and address offsetting techniques to improve cache performance

In cases of drastic increases in miss rates relative to a single-threaded architecture, the key to better performance probably lies in larger or more associative cache structures, page mapping and data layout techniques to reduce cache conflicts, or a combination of both. A previous study [7] examined OS page mapping and address offsetting strategies to use in conjunction with SMT. If avoidable cache conflicts are present due to artifacts of the traces (i.e., two traces' data segments overlap in the caches, say, though both would fit in the cache if properly offset), these techniques may help.

In order to study the effects of real-storage allocation policies (page mapping policies), we modified the Figaro simulator to include a model of address translation, allowing three separate page allocation policies. Four address translation policies were provided.

NODAT. No translation of the address in the trace.

COLOR. Addresses are allocated by page-coloring, which leaves the cache index bits unchanged by translation. This preserves spatial locality with respect to equivalence conflicts.

BINHOP. Pages are allocated sequentially per thread, so that temporal locality within a thread will tend to avoid cache equivalence conflicts.

RANDOM. Pages are allocated from a pseudo-random sequence (ignoring threads), wiping out both spatial and temporal locality effects.

In all cases, real storage was assumed to be unbounded, so that no page replacement took place.

3 Results

As previously noted, the present study was originally started a few years ago as an immediate follow-on to some of our previous research in the area of coarse-grained multithreading [4, 3]. Over this relatively long period of time, a very large number of experiments were performed with the Figaro simulator under a broad range of parameter settings, workloads and processor model configurations. The results of these simulation experiments have varied significantly, in absolute value, across this entire spectrum of configurations, exactly as one would expect. It is important to point out, however, that the relative trends of these simulation results have always remained consistent throughout this long sequence of experiments.

In this section we discuss a representative sample of our findings. Our primary focus is on the results obtained under technology assumptions and configuration settings that are consistent with those of previous studies in the area. This makes it possible for us to draw closer and more representative comparisons with the spectrum of previous work. In a similar manner, we also discuss results obtained from a version of the Figaro simulator that is configured to be as close to the SMT model in [12] as possible; see Section 3.6. Our on-going research further considers instances of the PowerPC-based processor model under assumptions based on current and next-generation technology so that additional comparisons can be drawn with the results of previous work, thus also supplementing earlier studies based on older technologies.

The primary performance measures presented are the average number of instructions executed per cycle (IPC) and the miss ratios of all caches. We compute IPC and other statistics for SMT simulations and compare to single-threaded performance as follows. In SMT mode our simulator halts when one trace runs out of instructions. Thus our SMT measurements reflect only multi-threaded performance. We record the position in the trace of the second thread, the one that did not run to completion, and extract the statistics for that initial portion of the trace from a single threaded run. The combined single-threaded IPC of the two traces is then computed by adding the total number of instructions executed and dividing by the total number of cycles on the two single-threaded runs (one complete and one partial). Thus we are comparing single-threaded and multi-threaded performance on exactly the same set of instructions.

3.1 Base configuration results

For all benchmarks except TPC-C, we obtained the best SMT results across all pairwise comparisons using the fair next-ready thread selection policy (note that this is slightly different from the results in Section 3.4 which are based on a subset of the pairwise comparisons). Thus we use this policy version of the SMT model defined in Section 2 as our base SMT configuration. Table 2 shows the percentage IPC improvement (which if negative indicates a degradation rather than an improvement) due to SMT, broken down in several ways. First, the results are shown for each benchmark when paired with all the other benchmarks. Among these pairings, the minimum, maximum, numbers of positive and negative improvements, and average are shown. After the per-benchmark statistics, the same data are shown for several aggregations: all pairs of two SPECint benchmarks, all pairs of two SPECfp benchmarks, all pairs of one SPECint benchmark and one SPECfp benchmark, and finally all pairs of benchmarks including the twelve SPEC benchmarks and TPC-C.

benchmarks	min	max	< 0	≥ 0	avg
compress	10.50	48.40	0	12	33.77
gcc	9.20	43.90	0	12	26.76
go	11.80	48.40	0	12	31.61
hydro2d	-2.00	16.10	2	10	8.72
ijpeg	9.90	46.00	0	12	29.32
li	6.40	45.30	0	12	26.35
m88ksim	7.90	42.50	0	12	26.68
mgrid	-5.60	24.70	3	9	12.54
perl	11.20	47.60	0	12	29.39
su2cor	3.40	26.70	0	12	17.18
swim	-5.60	18.80	3	9	8.17
tomcatv	-5.50	14.60	2	10	7.22
TPC-C	10.90	38.90	0	12	26.67
2 SPECint	32.20	48.40	0	21	39.42
2 SPECfp	-5.60	8.40	5	5	1.47
SPECint/fp	6.40	26.70	0	35	15.53
all pairs	-5.60	48.40	5	73	21.88

Table 2. Base configuration IPC percentage improvement summary

As can be seen in the table, we observe a substantial performance gain with integer applications including SPECint and TPC-C, but little or no gain and even degradation with SPECfp, and something in between with a combination. As will be seen, in the combination cases, the SPECfp application performs well and the integer application does not. We explore the reasons for these results in later sections.

Primary statistics affecting performance, of course, are cache miss rates. Tables 3 and 4 show the single-threaded cache and TLB miss statistics we measured for each benchmark and the increases in miss rates summarized over the SMT simulation runs, respectively. The measurements in

L1I	L1D	L2	ITLB	DTLB	TLB2
0.0002	3.8756	0.0327	0.0000	0.2244	0.0012
0.5062	0.2714	0.0107	0.0046	0.0229	0.0008
0.0833	1.8875	0.0047	0.0001	0.0008	0.0002
0.0016	1.7928	1.2500	0.0001	0.2891	0.0268
0.0031	0.1110	0.0320	0.0000	0.0029	0.0010
0.0045	0.3136	0.0032	0.0001	0.0002	0.0002
0.0017	0.0640	0.0162	0.0001	0.0023	0.0011
0.0017	0.9319	0.4743	0.0001	0.0534	0.0116
0.0005	0.0097	0.0014	0.0000	0.0012	0.0001
0.0009	2.6336	0.5298	0.0001	0.3012	0.0141
0.0006	1.3199	0.8194	0.0000	0.0843	0.0341
0.0002	5.0593	1.0255	0.0000	0.0684	0.0324
2.0321	1.7660	0.6979	0.3210	0.8607	0.3277

Table 3. Base configuration single-thread miss rates

measure	min	max	avg	global
L1I	-20.1	1128	65.9	20.4
L1D	8.9	1285	68.2	36.5
L2	< 0.001	42.00	8.02	9.54
ITLB	< 0.001	974	82.9	22.5
DTLB	20.8	78432	2472	280
TLB2	< 0.001	2097	59.4	20.0
mispred	-26.90	21.20	1.47	5.28
useless fetch	-63.9	-7.8	-37.4	-36.7

Table 4. Miss rate increase summary

Table 3 are for the 13 benchmarks in alphabetical order as in Table 2, whereas Table 4 shows summary statistics over all 78 pairings of the 13 benchmarks. While the TPC-C miss rates seem to be low, we note that they are consistent with those provided in [8]; the TPC-C trace used in our study is a slightly more recent version of the one used in [8], which was obtained with the same tool and in the same environment as our trace. Such low miss rates for TPC-C helps to further explain our somewhat pessimistic results.

Table 4 shows the increase in branch misprediction rate under SMT and the decrease in the number of useless instructions fetched. Although the misprediction rate increases slightly, useless instruction fetches are substantially decreased; a benefit of SMT is that speculation need not be as aggressive. Some of the miss rate increases seem alarming. However, this reflects only an insignificant miss rate increasing to a small or modest one. A more accurate measure of the increase, perhaps, is that indicated in the column labeled “global increase.” Here we report the increase in the total number of misses, over all 78 pairings of benchmarks, relative to the number of misses in the corresponding 156 single thread runs (12 runs each of each the 13 benchmarks). Though still substantial, these increases are much more manageable. Still, some indicate stress and the need for a larger or more associative structure. We will return to the miss rates later in the paper when we analyze the reasons for performance of particular pairings of benchmarks.

Vortex benchmark. We conducted several experiments to see if we could eliminate the problem with the TLB miss rates when simulating vortex and another application. With our base parameter settings, vortex (and its running mate) suffered increases under SMT ranging from 93% to 2200% in DTLB misses, and from 89% to 9000% in TLB2 misses. The SMT IPC decrease ranged from -2% to -78%; there were no increases. Doubling TLB sizes helped with the TLB2 misses, bringing the increases in miss rate down to a modest range of 10-50% and the worst IPC decrease to -38%; four pairings of benchmarks showed an IPC increase, the greatest being 11%. However, the increased size did not help with the level 1 DTLB misses. Increasing the DTLB associativity from 2 to 4 (without changing any sizes) was the key to relieving the problem, bringing the DTLB miss increases to 22-194%, the TLB2 miss increases to 11-89%, and the SMT IPC change to a range of -11 to $+12\%$ with an average of 4.6%.

3.2 Pipeline utilization

Turandot captures utilization statistics for several pipeline stages of interest. Figure 1 shows the utilization of the fetch, dispatch, and retire stages for single-threaded TPC-C in terms of the percentage of cycles on which a given number of instructions/operations passed through each of these stages. Figure 2 shows this information for TPC-C with two threads.

Clearly the fetch bandwidth is adequate. The large spike at 4 in the number of instructions dispatched is due to the upstream decode pipeline width of 4. To determine whether this is a bottleneck, we increased the decode width to 9 for a subset of our benchmarks – comp, gcc, su2cor, swim, and TPC-C – and only gcc’s single-threaded throughput increased significantly, from 1.08 to 1.17 IPC. On the 10 pairwise SMT runs between these 5 bechmarks similar results were obtained: we saw only a 1-2% increase in IPC relative to decode width 4 for those runs not involving gcc, and increases of 4-8.5% for those runs involving gcc.

With two threads, the number of cycles in which one instruction is retired decreases. This is because during the cycles in which one thread retires only one instruction, the second thread may also retire some instructions. This moves some of these cycles from retiring one instruction to the bars with more than one instruction retired. Also, as is to be expected, the number of cycles in which 8 or 9 instructions retire increases with two threads.

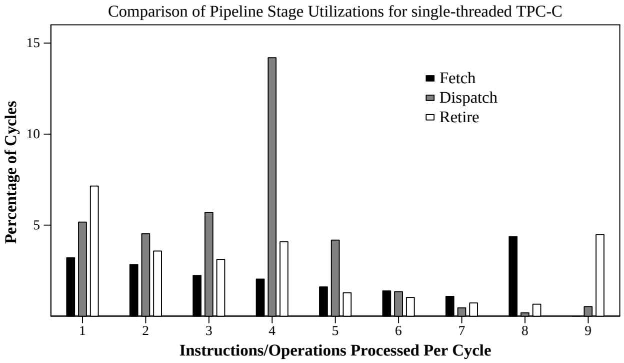

Figure 1. Utilization of pipeline stages for single-thread TPC-C.

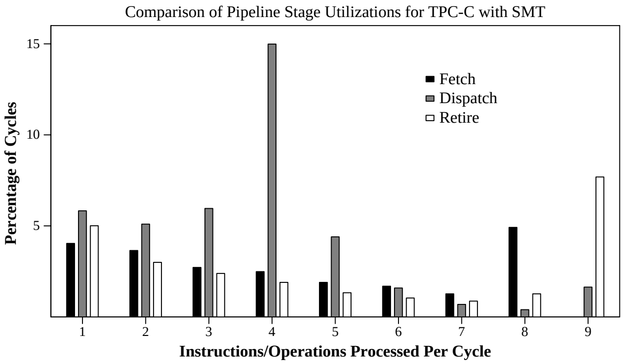

Figure 2. Utilization of pipeline stages for 2-thread TPC-C.

benchmarks	selfishness	std-dev	cv
su2cor	69.8	13.1	0.19
tomcatv	68.7	9.4	0.14
mgrid	67.5	13.7	0.20
hydro2d	64.2	11.1	0.17
perl	62.0	15.0	0.24
swim	61.6	12.5	0.20
comp	60.8	13.3	0.22
ijpeg	60.4	15.2	0.25
TPC-C	59.9	14.2	0.24
go	55.1	14.6	0.26
m88ksim	54.9	13.7	0.25
gcc	54.8	14.5	0.27
li	52.3	14.5	0.28
	citizenship	std-dev	cv
go	76.3	3.9	0.05
li	74.1	3.4	0.05
comp	73.0	8.9	0.12
gcc	71.8	4.5	0.06
m88ksim	71.8	4.0	0.06
ijpeg	68.9	4.6	0.07
perl	67.6	5.8	0.09
TPC-C	66.7	4.4	0.07
su2cor	47.2	6.4	0.13
swim	46.4	5.3	0.11
mgrid	45.2	5.6	0.12
hydro2d	44.4	8.0	0.18
tomcatv	38.6	5.7	0.15

Table 5. Benchmark citizenship

3.3 Analysis of thread citizenship

We define two qualities of an application, *selfishness* and *citizenship*. Selfishness is the relative speed of an application when running in SMT mode as measured by its IPC as a percentage of its single-threaded IPC. (A large value of selfishness is not entirely a bad thing despite the name; it means that the application performs well in SMT mode.) Citizenship is the fraction of single-threaded IPC that is achieved by the *other* application sharing the processor.

Table 5 shows the selfishness and citizenship measured for each of our benchmarks, averaged over all other benchmarks, along with the standard deviation (std-dev) and coefficient of variation (cv). The benchmarks are listed in order of decreasing selfishness and citizenship, respectively.

We make two observations regarding these results. First, citizenship has lower variation than selfishness, despite its much wider range of values. Thus, the application a2 that a given application a1 shares the processor with is a better predictor of a1's performance loss due to sharing than a1 itself. Second, floating point applications are bad citizens, whereas integer applications are good citizens.

As a possible explanation for this, we offer the following. Table 6 shows the single-thread L2 cache miss rates per 100 instructions and per 100 *cycles* by application, sorted in ascending order of the latter measure. Note that the order closely agrees with that of citizenship. Note also the sharp gap between the SPECfp and integer applications in both lists. We propose that overall performance is dominated (unsurprisingly) by memory performance, and that an application is forced to live with the memory penalty of an application it shares the processor with. We note that TPC-C is on the "wrong" side of the sharp drop in citizenship. That is, TPC-C is a better citizen than one would guess based on its miss rate. One explanation for this is TPC-C's high instruction cache miss rates. While data cache misses decrease citizenship, since they cause a thread's instructions to back up in the pipeline and consume resources and also cause cache pollution, instruction caches misses increase citizenship: the thread blocks and lets the other thread use the pipeline resources. This is borne out by simulations with infinite instruction and data caches: with an infinite data cache, TPC-C's citizenship increased to 69%, and with an infinite instruction cache, it decreased to 61%. We hypothesize that TPC-C's high citizenship is also due in part to the high prevalence of the PowerPC `isync` instruction, which blocks the thread and leaves all resources free for the other thread.

benchmarks	L2 misses (cycles)	L2 misses (instrs)
perl	0.0014	0.0014
go	0.0037	0.0047
li	0.0045	0.0032
gcc	0.0116	0.0107
m88ksim	0.0256	0.0162
comp	0.0291	0.0327
ijpeg	0.0491	0.0320
TPC-C	0.4589	0.6979
su2cor	0.5294	0.5298
mgrid	0.7390	0.4743
swim	1.2916	0.8194
tomcatv	1.2997	1.0255
hydro2d	1.4848	1.2500

Table 6. L2 misses per 100 cycles and per 100 instructions

3.4 Thread selection policy results

In Table 7 we report the results obtained by varying the thread selection mechanism as described in Section 2.3.3, where (F) and (U) respectively denote the fair and unfair versions of a given policy. Because it would require $78 \cdot 12 = 1248$ simulator runs otherwise, we report results only for a subset of the benchmarks: all pairs of compress, gcc, su2cor, swim, and TPC-C. Note that fair every cycle actually outperforms fair next ready in one case (compress and gcc), and has a slightly better average on the subset of pairwise comparisons considered in Table 7. This is in contrast to fair next ready providing the best SMT results across all pairwise comparisons with the exception of TPC-C. For one pair of benchmarks (su2cor and swim), the fair coarse grain policy results in one benchmark (su2cor) being starved, presumably because the other (swim) fits entirely in the instruction cache. We omit this pair in the statistics for this policy. These results tend to differ in various ways from the corresponding results presented in [12].

policy	min	max	< 0	≥ 0	avg
(F) branch count	-2.10	32.50	2	8	11.11
(F) every cycle	-2.20	37.20	1	9	17.29
(F) coarse grain	-5.80	6.50	4	5	0.09
(F) miss count	-6.30	30.70	1	9	11.89
(F) instr. count	-3.20	31.40	1	9	14.77
(F) next ready	-1.90	35.80	1	9	17.25
(F) op. count	-4.10	29.50	1	9	13.36
(F) priority	-3.30	20.20	1	9	8.31
(U) branch count	-2.20	32.50	1	9	11.19
(U) every cycle	-2.30	34.90	1	9	16.92
(U) coarse grain	-4.30	20.30	1	9	9.39
(U) miss count	-6.30	29.50	1	9	11.85
(U) instr. count	-3.10	31.70	1	9	15.01
(U) next ready	-2.20	33.30	1	9	16.50
(U) op. count	-3.90	29.80	1	9	13.66
(U) priority	-2.90	25.70	1	9	12.13

Table 7. Thread selection policy IPC percentage improvement summary

3.5 Page mapping results

Results of these experiments using the TPC-C trace, single- and multi-threaded, appear in Table 8. Case 1 represents the NODAT vs COLOR comparison, Case 2 represents the COLOR vs RANDOM comparison, and Case 3 represents the RANDOM vs BINHOP comparison. "Lines" denotes the number of cache lines transferred. COLOR and NODAT are nearly indistinguishable. BINHOP appears to be best in terms of throughput for two threads. RANDOM and BINHOP were identical for one thread. Though the miss rate reductions are impressive, the speedup in terms of IPC/CPI is modest: BINHOP is 1.7% faster than NODAT.

one thread	CPI	L2I / L2D misses	L2 lines
Case 1	+0.3%	+0.1% / +0.6%	+0.07%
Cases 2,3	+0.3%	+10.2% / +2.9%	+3.0%
two threads	**CPI**	**L2I / L2D misses**	**L2 lines**
Case 1	+0.1%	+0.2% / +4.7%	0%
Case 2	+1.3%	+13.7% / +8.0%	+8.7%
Case 3	+0.3%	+4.3% / +1.9%	+1.7%

Table 8. page mapping results (relative)

These results agree substantially with those reported in [7]: large effect on L2 miss rates, small effect on L1 miss rates (in the absence of address offsetting, a mechanism which we did not investigate at the time), except that we see a much smaller effect on IPC than they do. A partial explanation for this difference may be their 8-thread SMT vs our 2-thread: IPC results were only given for 8 threads, and miss rates for 8 threads show a much greater difference between COLOR and BINHOP than for 2 threads.

3.6 Analysis and comparison to previous results

The magnitude of our results are somewhat more modest than those of Tullsen et al. [12]. That study found a throughput increase of about 1/3 on average with two threads, using only a simple round-robin fetch policy as we do. We find a similar increase for SPECint benchmarks, but no improvement (and even a slight degradation) for SPECfp, a smaller increase for mixed SPECint and SPECfp, and a slightly smaller improvement for TPC-C combinations.

To analyze these issues in more detail, we parameterized our Figaro SMT model to be as close to the SMT model in [12] as possible. Our results did indeed show improved SMT performance, but we saw a corresponding improvement in the single-thread performance which maintained a comparable relative performance improvement for SMT to that shown above.

There are a number of possible explanations for these differences between our results and those presented in [12]. They include both architectural differences and workload differences, each of which are discussed in turn.

The PowerPC instruction set architecture contains complex instructions that are expanded into multiple simpler operations, each comparable to an Alpha instruction, at the decode stage. We parameterized our simulator to expand the instructions before they were fetched by the simulated processor to examine the effects of this difference. The results were very little changed, unsurprisingly; the bottlenecks appear to be further down the pipeline.

The major difference in cache organizations is that they assume an intermediate level of cache (which we would call L1.5, if we had it) with latency modestly greater (12 cycles vs. 6) than L1, and 256 kilobyte size. This cache is more than twice the combined sizes of our L1 instruction and data caches, and has only slightly greater latency.

They assume more (floating point) and more flexible (other) functional units: 3 FPUs compared to our 2, and 6 integer/branch units, 4 of which process loads and stores, compared to our 3 integer units, 2 load/store units, and 2 branch units. They also consider a somewhat longer pipeline. Another architectural difference concerns the register file size. They assume 32 registers per thread plus 100 excess registers for renaming (for each of the floating point and integer register files); we assume 128 registers regardless of the number of threads. With respect to the decode bandwidth, they assume a bandwidth of 8 whereas ours is 4.

In terms of workload differences, they use SPEC92 benchmarks whereas we use SPEC95 and TPC-C. (A subsequent paper [7] reports on simulations using an OLTP workload, but reports only results for eight threads.) Moreover, these benchmarks were obtained under different compilers (Multiflow vs. xlc/xlf). Our results in Section 3.5 suggest that this may have an important impact on the differences between our results. Finally, the larger cache footprints of our workload (see http://www.specbench.org/osg/cpu95/whatsnew.html) increase cache contention, possibly reducing the SMT performance gain.

We also believe our current simulation results to be somewhat pessimistic with respect to the potential performance benefits of SMT. There are several reasons for this. First, the Turandot simulator does not model in complete detail some sources of latency that cause pipeline underutilization, e.g., bus conflicts. This means that our single-threaded results tend to be somewhat better than reality, and thus there is less latency for SMT to hide. Second, our handling of mispredicted branch instructions in Figaro uses the pessimistic approach of keeping purged instructions in the pipeline, consuming resources, until retirement; see Section 2.3.2. Furthermore, there are many opportunities for optimization to take full advantage of SMT. As previously noted, one of our objectives was to explore the benefits and limitations of SMT within the context of an existing PowerPC-based wide superscalar processor model that is not optimized in any major way to maximize SMT performance. There are a number of techniques that could be employed in this minimally modified PowerPC-based processor to achieve the maximum performance improvements that may be possible with SMT.

One such area of optimization concerns split versus shared structures in the SMT architecture. Several queue structures, perhaps most importantly the instruction buffer (into which instructions are fetched from the cache and from which they move to the decoder), can be designed either as a single queue shared by the threads or split into separate structures, one for each thread. The tradeoff here is between flexibility in the number of entries that can be dedicated to a single thread (all of them in the case of a shared structure, but only half of them in the case of a split structure) versus the ability to select which thread's instructions to process (possible in the case of a split structure, not possible in the case of a shared structure). The queues are shared in our current simulator. A possible enhancement to the simulator would allow split structures in order to examine this tradeoff. Because instructions are dispatched in fetch order, a downstream stall in one thread can stall the other thread's dispatch if there is a full rename, or similar, queue.

Many other optimizations are possible. This includes increasing the number of threads, which would do a better job of hiding the larger latencies that might be found with current and next-generation technology, but at the expense of greater chip area. A longer pipeline would also show larger relative benefits for SMT. We are exploring some of these issues as part of on-going research.

4 Concluding Remarks

In this paper we presented a performance analysis of SMT within the context of a PowerPC-based wide superscalar processor architecture under a broad range of workloads, which included combinations of TPC-C, SPECint and SPECfp. There are several important differences between this PowerPC-based architecture and the processor architectures of previous studies, and we explored the impact of some of these differences on SMT design and performance issues. Although some of our results are consistent with previous work, our results also demonstrate some differences which we investigated to identify the primary causes of such differences. This includes an investigation of thread characteristics that work well together in SMT environments, thus providing significant performance benefits, and thread characteristics that do not work well together, thus providing smaller improvements and in some cases resulting in performance degradation.

Acknowledgements

We thank our colleagues Hal Kossman, Jaime Moreno, Mayan Moudgill, Mike Wazlowski and Eric Wu for many fruitful discussions regarding various aspects of this research. We further thank Mayan Moudgill for his help and assistance in our modifications of the Turandot simulator. We also thank the anonymous referees for helpful comments on an earlier draft of this paper. Finally, we thank Jerry Bozman for suggesting Mozart's "Le Nozze di Figaro" (The Marriage of Figaro) for the name of our SMT simulator.

References

[1] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[2] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A foundation for next-generation processors. *IEEE Micro*, October 1997.

[3] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, B.-H. Lim, M. S. Squillante, and C. E. Wu. Evaluation of multithreaded processors and thread-switch policies. In *Proceedings of the International Symposium on High Performance Computing*, November 1997.

[4] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[5] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the Second International Symposium on High-Performance Computing*, pages pages 291–301, February 1996.

[6] Intel Corporation. Intel Xeon processor family for servers with hyper-threading technology: Offering increased server performance through on-processor thread-level parallelism. White Paper, 2002.

[7] J. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[8] A. M. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectu ral Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[9] M. Moudgill, P. Bose, and J. Moreno. Validation of Turandot, a fast processor model for microarchitecture exploration. In *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 452–457, February 1999.

[10] M. Moudgill, J.-D. Wellman, and J. Moreno. Environment for PowerPC microarchitecture exploration. *IEEE MICRO*, pages 15–25, May/June 1999.

[11] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[12] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[13] D. M. Tullsen, S. J. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[14] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[15] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.