Software-Controlled Priority Characterization of POWER5 Processor

Carlos Boneti † Francisco J. Cazorla ⋆ Roberto Gioiosa ⋆
Alper Buyuktosunoglu‡ Chen-Yong Cher ‡ Mateo Valero †⋆

† Universitat Politècnica de Catalunya, Spain {cboneti, mateo}@ac.upc.edu
⋆ Barcelona Supercomputing Center, Spain {francisco.cazorla, roberto.gioiosa, mateo.valero}@bsc.es
‡ IBM T. J. Watson Research Center, Yorktown Heights, USA {alperb, chenyong}@us.ibm.com

Abstract

Due to the limitations of instruction-level parallelism, thread-level parallelism has become a popular way to improve processor performance. One example is the IBM POWER5™ processor, a two-context simultaneous-multithreaded dual-core chip. In each SMT core, the IBM POWER5 features two levels of thread resource balancing and prioritization. The first level provides automatic in-hardware resource balancing, while the second level is a software-controlled priority mechanism that presents eight levels of thread priorities. Currently, software-controlled prioritization is only used in limited number of cases in the software platforms due to lack of performance characterization of the effects of this mechanism.

In this work, we characterize the effects of the software-based prioritization on several different workloads. We show that the impact of the prioritization significantly depends on the workloads coscheduled on a core. By prioritizing the right task, it is possible to obtain more than two times of throughput improvement for synthetic workloads compared to the baseline. We also present two application case studies targeting two different performance metrics: the first case study improves overall throughput by 23.7% and the second case study reduces the total execution time by 9.3%. In addition, we show the circumstances when a background thread can be run transparently without effecting the performance of the foreground thread.

1 Introduction

The limitations imposed by the Instruction Level Parallelism (ILP) have motivated the use of Thread-level parallelism (TLP) as a common strategy to improve processor performance. Common TLP paradigms are Simultaneous Multi-Threading (SMT) [20][23], Chip-Multi-Processor (CMP) [1][13], or combinations of both [21].

The IBM POWER5 is a dual-core processor, where each core runs two threads. Threads share many resources such as the Global Completion Table (GCT or reorder buffer), the Branch History Table (BHT) and the Translation Lookaside Buffer (TLB). It is well known that higher performance is realized when resources are appropriately balanced across threads [16][17][21]. An IBM POWER5 system appropriately balances the usage of resources across threads with mechanisms in hardware [10][16]. Moreover, POWER5 employs a mechanism, through software/hardware co-design, that controls the instruction decode rate with eight priority levels. Its main motivation is to address instances where unbalanced thread priority is desirable. Several examples can be enumerated such as idle thread, thread waiting on a spin-lock, software determined non-uniform balance and power management [17][21]. Software-controlled priority[1] can significantly improve both throughput and execution time depending on the application type.

In the literature, a wide range of mechanisms has been proposed on dynamically balancing of resources to improve SMT performance. Most of these proposals focus on the instruction fetch policy as the means to obtain such balancing [8][22]. In addition to the instruction fetch policy, other mechanisms explicitly prioritize shared resources among threads to improve throughput, fairness [2][6] and Quality of Service [4]. While these studies do not correspond to the software prioritization mechanism of POWER5, they could justify the use of the mechanism.

Nevertheless, the prioritization mechanism provided by POWER5 is rarely used among the software community and, even in these rare cases, the prioritization mechanism is mainly used for lowering the priority of a thread. For instance, the current Linux kernel (version 2.6.23) exploits the software-controlled priorities in few cases to reduce the priority of a processor that is not performing any useful computation. Moreover, Linux makes the assumption that the software-controlled priority mechanism is not used by the programmer and resets the priority to the default value at every interrupt or exception handling point.

Currently, the lack of quantitative studies on software-controlled priority limit their use in real world applications. In this paper, we provide the first quantitative study of the

[1] POWER5 software-controlled priorities are independent of the operating systems concept of process or task priorities.

1063-6897/08 $25.00 © 2008 IEEE
DOI 10.1109/ISCA.2008.8

POWER5 prioritization mechanism. We show that the effect of thread prioritization depends on the characteristics of a given thread and the other thread it is coscheduled with. Our results show that, if used properly, software-controlled priorities may increase overall system performance, depending on the metric of interest. Furthermore, this study helps Linux and other software communities to tune the performance of their software by exploiting the software-controlled priority mechanism of the POWER5 processor.

The main contributions of this paper are:

1. We provide a detailed analysis of the effect of the POWER5 prioritization mechanism on execution time of applications with a set of selected micro-benchmarks that stress specific workload characteristics. We show that:

- Threads executing long-latency operations (i.e., threads with a lot of misses in the caches) are less effected by priorities than threads executing short-latency operations (i.e. cpu-bound threads). For example, we observe that increasing the priority of a cpu-bound thread could reduce its execution time by 2.5x over the baseline. Increasing the priority of memory-bound threads causes an execution time reduction of 1.7x when they are run with other memory-bound threads.

- By reducing the priority of a cpu-bound thread, its performance can decrease up to 42x when running with a memory-bound thread and up to 20x when running with another cpu-bound thread. In general, improving the performance of one thread involves a higher performance loss on the other thread, sometimes by an order of magnitude. However, decreasing the priority of a long-latency thread has less effect on its execution time compared to a cpu-bound thread. For example, decreasing the priority of a memory-bound thread increases its execution time by 22x when running with another memory-bound thread, while increases less than 2.5x when running with the other benchmarks. In Section 5.3 we show how to exploit this to improve the overall performance.

- For the micro-benchmarks used in this paper, the IPC throughput of the POWER5 improves up to 2x by using software-controlled priorities.

- We also show that a thread can run transparently, with almost no impact on the performance of a higher-priority thread. In general, foreground threads with lower IPC are less sensitive to a transparent thread.

2. We present two application case studies that show how priorities in POWER5 can be used to improve two different metrics: aggregated IPC and execution time.

- In the case of a batch application where the main metric is throughput, the performance improves up to 23.7%.

- In the case of an unbalanced MPI parallel application, execution time reduces up to 9.3% by using priorities to re-balance its resources.

To our knowledge, this is the first quantitative study showing how software-controlled prioritization of POWER5 effects performance on a real system. Since other processors like the IBM POWER6™ [12] present a similar prioritization mechanism, this study can be significantly useful for the software community.

This paper is organized as follows: Section 2 presents the related work. Section 3 describes the POWER5 resource balancing in hardware and the software-controlled priority mechanisms. Section 4 presents our evaluation environment, and Section 5 shows our results and their analysis. Finally, Section 6 concludes this work.

2 Related Work

In the literature a wide range of mechanisms have been proposed to prioritize the execution of a thread in a SMT processor. Many of these proposals focus on the instruction-fetch policy to improve performance and fairness in SMT processors, while other focus on explicitly assigning processor resources to threads.

Instruction Fetch Policies: An instruction fetch (I-fetch) policy decides how instructions are fetched from the threads, thereby implicitly determining the way processor resources, like rename registers or issue queue entries, are allocated to the threads. Many existing fetch policies attempt to maximize throughput and fairness by reducing the priority, stalling, or flushing threads that experience long latency memory operations [2][22]. Some other fetch policies focus on reducing the effects of mis-speculation by stalling on hard-to-predict branches [18][19].

Explicit Resource Allocation: Some of the mechanisms explicitly allocate shared processor resources targeting throughput improvements [2][6]. Other resource allocation mechanisms provide better QoS guarantees for the execution time by ensuring a minimum performance for the time critical threads [3][4].

3 The POWER5 Processor

IBM POWER5 [15] processor is a dual-core chip where each core runs two threads [17]. POWER5 employs two levels of control among threads, through resource balancing in hardware (Section 3.1), as well as software-controlled prioritization (Section 3.2).

3.1 Dynamic hardware resource balancing

POWER5 provides a dynamic resource-balancing mechanism that monitors processor resources to determine whether one thread is potentially blocking the other thread execution. Under that condition, the progress of the offending thread is throttled back, allowing the sibling thread to progress. POWER5 considers that there is an unbalanced use of resources when a thread reaches a threshold of L2 cache or TLB misses, or when a thread uses too many GCT (reorder buffer) entries.

POWER5 employs one of the following mechanisms to re-balance resources among threads: 1) It stops instruction decoding of the offending thread until the congestion clears (Stall). 2) It flushes all of the instructions of the offending thread that are waiting for dispatch and stopping the thread from decoding additional instructions until the congestion clears (Flush). Moreover, the hardware may temporarily adjust the decode rate of a thread to throttle its execution.

3.2 Software-controlled priorities

The number of decode cycles assigned to each thread depends on the software-controlled priority. The enforcement of these software-controlled priorities is carried by hardware in the decode stage. In general, the higher the priority, the higher the number of decode cycles assigned to the thread.

Let us assume that two threads, a primary thread (*PThread*) and a secondary thread (*SThread*), are running on one of the two cores of the POWER5 with priorities PrioP and PrioS, respectively. Based on the priorities, the decode slots are allocated using the following formula:

$$R = 2^{|PrioP - PrioS| + 1} \quad (1)$$

Notice that R is computed using the difference of priorities of *PThread* and *SThread*, *PrioP-PrioS*. At any given moment, the thread with higher priority receives R-1 decode slots, while the lower priority thread receives the remaining slot. For instance, assuming that *PThread* has priority 6 and *SThread* has priority 2, R would be 32, so the core decodes 31 times from *PThread* and once from *SThread* (more details on the hardware implementation are provided in [10]). The performance of the process running as *PThread* increases to the detriment of the one running as *SThread*. On the special case where both threads have the same priority, $R = 2$, and therefore, each thread receives one slot, alternately.

In POWER5, the software-controlled priorities range from 0 to 7, where 0 means the thread is switched off and 7 means the thread is running in Single Thread (ST) mode (i.e., the other thread is off). The supervisor or operating system can set six of the eight priorities ranging from 1 to 6, while user software can only set priority 2, 3 and 4. The Hypervisor can always use the whole range of priorities.

As described in [10] and [15], the priorities can be set by issuing an `or` instruction in the form of `or X,X,X`, where `X` is a specific register number. This operation only changes the thread priority and performs no other operation. If it is not supported (when running on previous POWER processors) or not permitted due to insufficient privileges, the instruction is simply treated as a `nop`. Table 1 shows the priorities, the privilege level required to set each priority and how to change priority using this interface.

Table 1. Software-controlled thread priorities in the IBM POWER5 processor.

Priority	Priority level	Privilege level	or-nop inst.
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User/Supervisor	`or 1,1,1`
3	Medium-Low	User/Supervisor	`or 6,6,6`
4	Medium	User/Supervisor	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

The behavior of the software-controlled thread prioritization mechanism is different when one of the threads has priorities 0 or 1 as shown in [10][15]. For instance, when both threads have priority one, instead of being considered as difference 0 and perform as having no prioritization, the processor runs in low-power mode, decoding only one instruction every 32 cycles.

4 Evaluation Methodology

In order to explore the capabilities of the software-controlled priority mechanism in the POWER5 processor, we performed a detailed set of experiments. Our approach consists of analyzing the processor as a black-box, observing how the performance of a workload changes as we increase or reduce the priority of threads.

In a SMT processor the performance of one process not only depends on the processor architecture but also on the other processes running at the same time on the same core and their specific program phases. Under such conditions, evaluating all the possible programs and all their phase combinations is simply not feasible. Moreover, when it comes to a real system evaluation, with the several layers of the running software, the OS interferences and all the asynchronous I/O services, the problem becomes even worse.

For this reason, we use a set of micro-benchmarks that stresses a particular processor characteristic. While this scenario is not typical with real applications, this is one of the best ways to understand the mechanism under evaluation. It

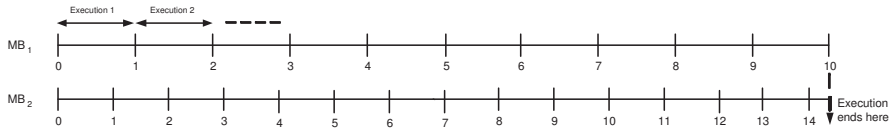


Figure 1. Example of application of the FAME methodology. In this example Micro-Benchmark 1 takes longer than Micro-Benchmark 2.

provides a uniform characterization based on specific program characteristics that can be mapped into real applications. With real applications it would be impossible to attribute fine-grain performance gain/loss to the prioritization mechanism due to applications own variability.

4.1 Running the experiments

This paper uses the FAME (FAirly MEasuring Multithreaded Architectures) methodology [24][25]. In [24] the authors state that the average accumulated IPC of a program is representative if it is similar to the IPC of that program when the workload reaches a steady state. The problem is that, as shown in [24][25], the workload has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value.

For the experimental setup and micro-benchmarks used in this paper, in order to accomplish a MAIV of 1%, each micro-benchmark must be repeated at least 10 times. In our experiments we run two workloads, hence each experiment ends when both threads re-execute at least 10 times. Note that, at this point the fastest thread might already execute more than 10 times. Figure 1 shows an example where the second benchmark is faster than the first. In this example, while the MB_1 ($MicroBenchmark_1$) executes 10 times, MB_2 executes 14 times. It is important to note that the execution time difference is not constant. For instance, if we change the software-controlled priorities, MB_2 may execute faster or slower, and therefore we must guarantee that both threads execute a minimum number of repetitions. In our experiments, the average execution time for a thread is estimated as the total accounted execution time divided by the number of *complete* repetitions. For example, in Figure 1, the total execution time of MB_2 is measured until it completes the 14th iteration and the time for the remaining incomplete iteration is discarded.

Furthermore, as previously shown [9][11], normal software environment can insert significant noise into performance measurements. To minimize such noise, both single-thread and multithreaded experiments were performed on the second core of the POWER5. All user-land processes and interrupt requests (IRQ) were isolated on the first one, leaving the second core as free as possible from noise.

4.2 Micro-benchmark

In a multi-threaded architecture, the performance of one process tightly depends on the other process that it is running with. Moreover, the effect of the software-controlled priorities depends on the characteristics of the benchmarks under study. In order to build a basic knowledge of these effects, we developed a set of 15 synthetic micro-benchmarks, each of them stressing a specific processor characteristic. This methodology allows us to isolate independent behaviors of the platform. Furthermore, micro-benchmarks provide higher flexibility due to their simplicity.

We classify the micro-benchmarks in four groups: Integer, Floating Point, Memory and Branch, as shown in Table 2. In each group, there are benchmarks with different instruction latency. For example, in the Integer group there are short (`cpu_int`) and long (`lng_chain_cpuint`) latency operation benchmarks. In the Memory group, `ldint_l2` is a benchmark with all loads always hitting in second level of data cache, while `ldint_mem` has all loads hitting in memory and, hence, missing in all cache levels. As expected, `ldint_l2` has higher IPC than `ldint_mem`. In the Branch group, there are two micro-benchmarks with high (`br_hit`) and low (`br_miss`) hit prediction rate.

All the micro-benchmarks have the same structure. They iterate several times on their loop body and the loop body is what differentiates them. One execution of the loop body is called a micro-iteration. Table 2 shows the details of the loop body structures for the micro-benchmarks. The size of the loop body and the number of micro-iterations is specific for each benchmark. The benchmarks are compiled with the `xlc` compiler with `-O2` option and their object code are verified in order to guarantee that the benchmarks retain the desired characteristics.

After the first complete set of experiments, where we ran all the possible combinations, we realized that some of the benchmarks behave equally and do not add any further insight to the analysis. Therefore, we present only the benchmarks that provide differentiation for this characterization

Table 2. Loop body of the different micro-benchmarks.

Name	Loop Body
cpu_int	a += (iter * (iter - 1)) - x_i * iter : $x_i \in \{1, 2, ..., 54\}$
cpu_int_add	a += (iter + (iterp)) - x_i + iter : $x_i \in \{1, 2, ..., 54\}$; iterp = iter -1 + a
cpu_int_mul	a = (iter * iter) * x_i * iter : $x_i \in \{1, 2, ..., 54\}$;
lng_chain_cpuint	a += (iter * (iter - 1)) - x_0 * iter : $x_i \in \{1, 2, ..., 20\}$ b += (iter * (iter - 1)) - x_1 * iter + a ... a += (iter + (iter - 1)) - x_{10} * j ... The cycle of instructions is repeated multiple times, for a total of 50 lines in the loop body.
br_hit br_miss	if (a[s]=0) a=a+1; else a=a-1; $s \in \{1, 2, ..., 28\}$ a is filled with all 0's for br_hit and randomly (modulo 2) for br_miss
ldint_l1 ldint_l2 ldint_l3 ldint_mem ldfp_l1 ldfp_l2 ldfp_l3 ldfp_lmem	a[i+s] = a[i+s]+1; where s is set that we always hit in the desired cache level. In the case of fp benchmarks, a is an array of floats.
cpu_fp	a += (tmp * (tmp - 1.0)) - x_i * tmp —$x_i \in \{1.0, 2.0, ..., 54.0\}$. (float tmp = iter * 1.0)

work. For example, br_hit, br_miss, cpu_int_add, cpu_int_mul and cpu_int behave in a very similar way. Analogously, the load-integers and load-floating-points do not significantly differ. Therefore, we present only the results for cpu_int, lng_chain_cpuint, ldint_l1, ldint_l2, ldint_mem and cpu_fp.

4.3 The Linux kernel

Some of the priority levels are not available in user mode (Section 3.2). In fact, only three levels out of eight can be used by user mode applications, the others are only available to the OS or the Hypervisor. Modern Linux kernels (2.6.23) running on IBM POWER5 processors exploit software-controlled priorities in few cases such as reducing the priority of a process when it is not performing useful computation. Basically, it makes use of the thread priorities in three cases:

- The processor is spinning for a lock in kernel mode. In this case the priority of the spinning process is reduced.
- The kernel is waiting for operations to complete. For example, when the kernel requests a specific CPU to perform an operation by means of a smp_call_function() and it can not proceed until the operation completes. Under this condition, the priority of the thread is reduced.
- The kernel is running the idle process because there is no other process ready to run. In this case the kernel reduces the priority of the idle thread and eventually puts the core in Single Thread (ST) mode.

In all these cases the kernel reduces the priority of a processor's context and restores it to MEDIUM (4) as soon as there is some job to perform. Furthermore, since the kernel does not keep track of the actual priority, and to ensure responsiveness, it also resets the thread priority to MEDIUM every time it enters a kernel service routine (for instance, an interrupt, an exception handler, or a system call). This is a conservative choice induced by the fact that it is not clear how and when to prioritize a processor context and what the effect of that prioritization is.

In order to explore the entire priority range, we developed a non-intrusive kernel patch which provides an interface to the user to set all the possible priorities available in kernel mode:

- We make priority 1 to 6 available to the user. As mentioned in Section 3.2, only three of the priorities (4, 3, 2) are directly available to the user. Without our kernel patch, any attempt to use other priorities result in a nop operation. Priority 0 and 7 (context off and single thread mode, respectively) are also available to the user through a Hypervisor call.
- We remove the use of software-controlled priorities inside the Linux kernel, otherwise the experiments would be effected by unpredictable priority changes.
- Finally, we provide an interface through the /sys pseudo file system which allows user applications to change their priority.

5 Analysis of the Results

In this section, we show to what extent the prioritization mechanism of POWER5 effects the execution of a given thread and the trade-off between prioritization and throughput.

The following sections use the same terminology as Section 3.2. We call the first thread in the pair the "Primary Thread", or *PThread*, and the second thread the "Secondary Thread" or *SThread*. The term PrioP refers to priority of the primary thread, while PrioS refers to the priority of the secondary thread. The priority difference (often expressed as

Table 3. IPC of micro-benchmarks in ST mode and in SMT with priorities (4,4). *pt* stands for *PThread* and *tt* for total IPC.

Micro benchmark	IPC ST	IPC in SMT (4,4)											
		ldint_l1		ldint_l2		ldint_mem		cpu_int		cpu_fp		lng_chain_cpuint	
		pt	tt	pt	tt	pt	tt	pt	tt	pt	tt	pt	tt
ldint_l1	2.29	1.15	2.31	0.60	0.87	0.79	0.81	0.73	1.57	0.77	1.18	0.42	0.91
ldint_l2	0.27	0.27	0.87	0.11	0.22	0.17	0.19	0.27	0.87	0.25	0.65	0.27	0.72
ldint_mem	0.02	0.02	0.81	0.02	0.19	0.01	0.02	0.02	0.90	0.02	0.39	0.02	0.48
cpu_int	1.14	0.84	1.57	0.59	0.87	0.88	0.90	0.61	1.22	0.65	1.06	0.43	0.86
cpu_fp	0.41	0.41	1.18	0.39	0.65	0.37	0.39	0.40	1.06	0.36	0.72	0.37	0.85
lng_chain_cpuint	0.51	0.49	0.91	0.45	0.73	0.47	0.48	0.43	0.86	0.48	0.85	0.42	0.85

$PrioP - PrioS$) can be positive in which case the *PThread* has higher priority than the *SThread* or negative where the *SThread* has higher priority. The results are normalized to the default case with priorities (4,4).

Table 3 presents the IPC values in single thread mode as well as in SMT mode with priorities (4,4). For each row, the column *pt* shows the IPC of the primary thread and *tt* shows the total IPC. For example, the second row presents the case where ldint_l2 is the primary thread. The IPC ST column shows its single thread IPC value (0.27). The third column present its IPC when running with ldint_l1 (0.27) and the fourth column shows the combined IPC of ldint_l1 and ldint_l2 when running together (0.87).

In the Sections 5.1 and 5.2 we discuss about the effects of negative and positive prioritization. This effect is not symmetric as it follows the formula 1. For instance, at priority +4 a thread receive 31 of each 32 decode slots, which represents an increase of 93.75% of the resources when compared to the baseline, where a thread receives half of the resources. However, at priority -4, a thread receives only one out of 32 decode slots, which represents 16 times less resources.

On the Figures 2 and 3, the results represent the relative performance of the primary thread shown in the graph's caption when coscheduled with each one of the other benchmarks in the legend. The results are a factor of the baseline case with no prioritization.

5.1 Effect of Positive Priorities

In this section, we analyze the performance improvement of the *PThread* with different *SThreads* using positive priorities (*PrioP* $>$ *PrioS*). Figure 2 shows the performance improvement of the *PThread* as its priority increases with respect to the *SThread*. For example, Figure 2 (c) shows the results when we run cpu_int as *PThread*.

In general, the threads that have low IPC and are not memory-bound, such as lng_chain_cpuint and cpu_fp, benefit less from having high priorities. Memory-bound benchmarks, such as ldint_l2 and ldint_mem, benefit from the prioritization mechanism when they run with another memory-bound thread. This improves performance up to 240% for ldint_l2 and 70% for ldint_mem. On the other hand, high IPC threads, like cpu_int and ldint_l1 are very sensitive to the prioritization as they can benefit from the additional hardware resources. Therefore, their prioritization usually improves the total throughput and increases their performance.

The results show that the memory-bound benchmarks are also effected by the POWER5 prioritization mechanism, when competing with other benchmarks of similar requirements. They are less sensitive than the purely cpu-bound benchmarks, and they only benefit from an increased priority when co-scheduled with other memory-bound benchmarks. As a rule of thumb, memory-bound benchmarks should not be prioritized except when running with other memory-bound benchmark. Section 5.3 shows that prioritizing these workloads is often in detriment of the overall system performance.

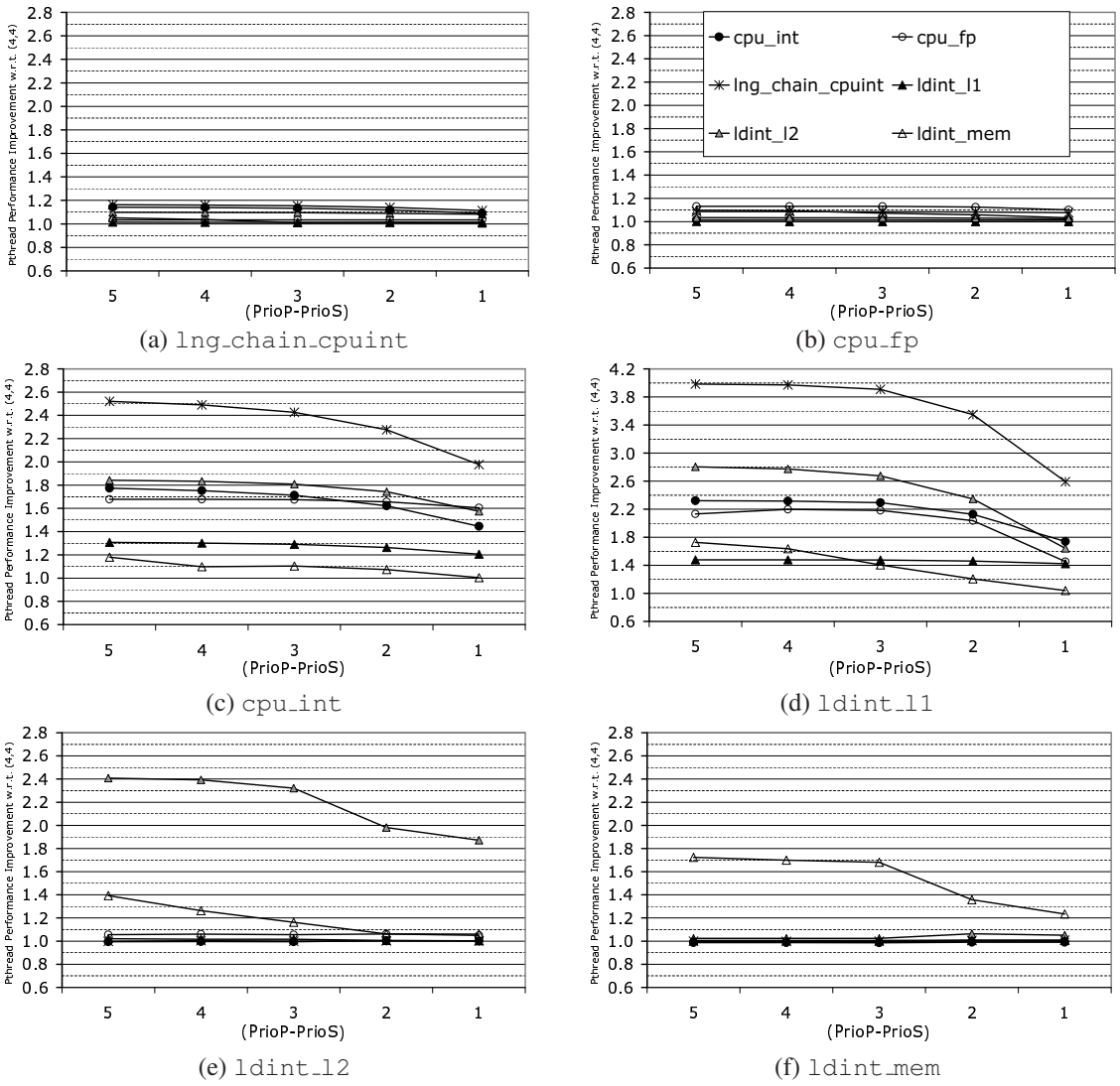
In addition, a priority difference of +2 usually represents a point of relative saturation, where most of the benchmarks reach at least 95% of their maximum performance. The memory-bound benchmarks represent an exception to this rule, where the largest performance increase occurs from a priority difference of +2 to +3.

5.2 Effect of Negative Priorities

In this section, we present the effects of the negative priorities (*PrioP* $<$ *PrioS*) on the micro-benchmarks. Figures 3 (a) to (e) show that setting negative priorities heavily impacts the performance of all micro-benchmarks except for ldint_mem. The effect of the negative priorities on the performance is much higher than the effect of the positive priorities. While using positive priorities could improve performance up to four times, negative priorities can degrade performance by more than forty times.

Figure 3 (f) presents that ldint_mem is insensitive to low priorities in all cases other than running with another thread of ldint_mem. In general, a thread presenting high latency memory operation, long dependency chains or slow complex operations is less effected by a priority reduction.

Memory-bound benchmarks are the ones that impact the other threads the most. They also present clear steps of per-



(a) `lng_chain_cpuint` (b) `cpu_fp`

(c) `cpu_int` (d) `ldint_l1`

(e) `ldint_l2` (f) `ldint_mem`

Figure 2. Performance improvement of the *PThread* as its priority increases with respect to the *SThread*. Note the different scale for `ldint_l1`.

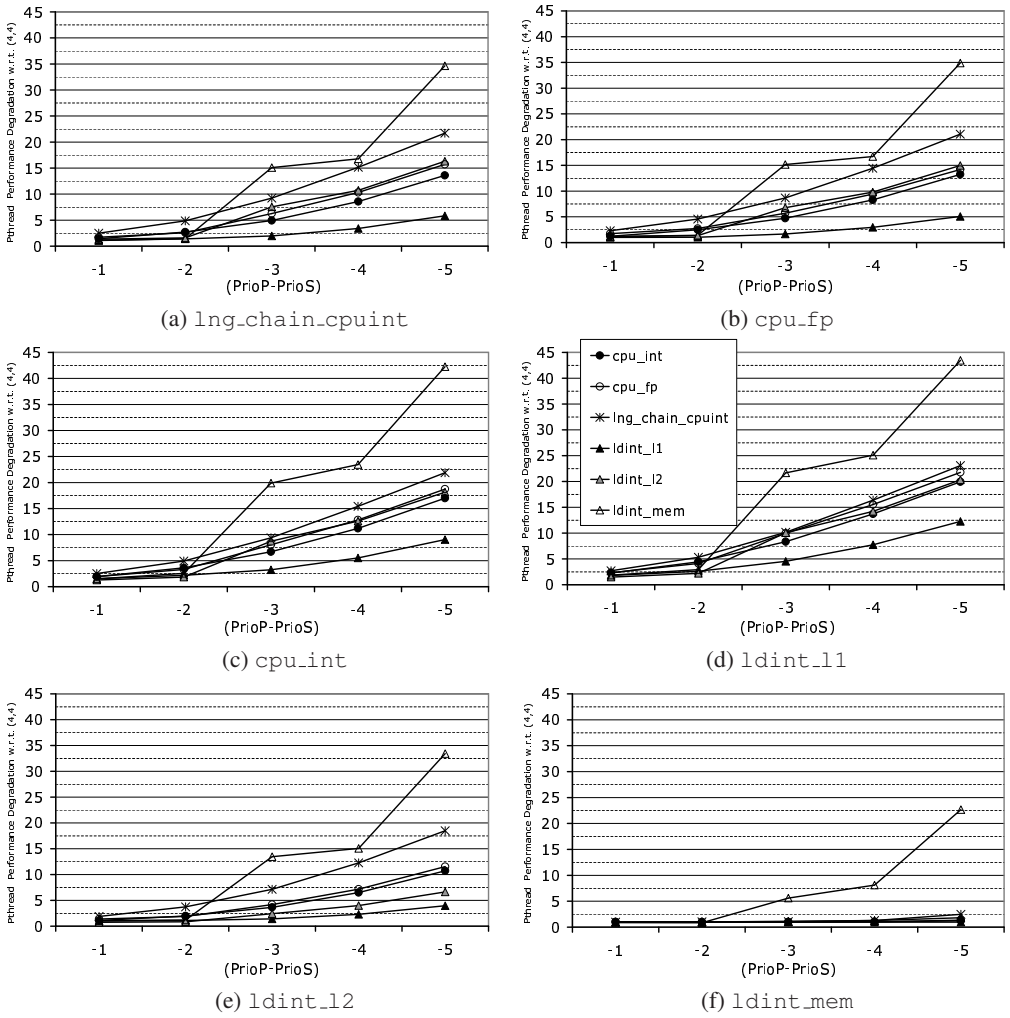
formance impact when the priority difference changes from -2 to -3, and from -4 to -5. The priority difference of -5 is extreme since the *PThread* obtains only the left-overs from the memory thread. In general, a priority difference like -5 should only be used for a transparent background thread in which the performance is not important.

While priority difference of +2 usually yields close to the maximum performance, priority -3 results in a clear delta on performance loss. For memory-bound threads, there is no significant performance variation from 0 to -2. Considering that priority difference +2, most of the high IPC threads reach 95% of their maximum performance, this suggests that priority differences larger than +/-2 should normally be avoided. Section 5.3 shows the additional throughput that can be obtained based on this conclusion.

5.3 Optimizing IPC Throughput

POWER5 employs several hardware mechanisms to improve the global throughput, like stalling the decode of the low IPC tasks or flushing the dispatch of threads that would otherwise decrease the overall performance of the system. The POWER5 built-in resource balancing mechanism is effective in most cases where changing the thread's priorities negatively impact the total throughput.

Even though the baseline is effective, Figure 4 shows several cases where the total throughput can be improved up to two times or more. This comes at the expense of severe slowdown of the low priority thread, espe-



(a) `lng_chain_cpuint` (b) `cpu_fp`

(c) `cpu_int` (d) `ldint_l1`

(e) `ldint_l2` (f) `ldint_mem`

Figure 3. Performance degradation of the *PThread* as its priority decreases with respect to the *SThread*.

cially when the low priority thread has low IPC such as `lng_chain_cpuint`. These cases can be exploited for systems where total throughput is the main goal and where the low IPC thread can actually afford the slowdown.

Furthermore, while the performance for the cpu-bound benchmarks increase with their priority, the performance of a memory benchmark remains relatively constant. Using the prioritization mechanism for this combination yields, almost always, significant throughput improvement. In general, we obtain an IPC throughput improvement when we increase the priority of the higher IPC thread in the pair.

5.3.1 Case Study

In order to verify whether our findings can be applied to real workloads, this section shows how software-controlled prioritization can improve total IPC. We analyze the behavior of two pairs of SPEC CPU 2000 and 2006 benchmarks [14]. The first one is composed of *464.h264ref* (from now on referred as *h264ref*) and *429.mcf* (from now on referred as *mcf*). The second pair is composed of *173.applu* (from now on referred as *applu*) and *183.equake* (from now on referred as *equake*). We take as the baseline the scenario where they run side by side, on different contexts of the same core, without any type of software-controlled prioritization (i.e., with the same priority). The experiments follow the FAME methodology.

When running with the default priorities (4,4), *h264ref* has an IPC of 0.920 and takes about 3254 seconds to complete, and *mcf* takes 1848 seconds and reaches an IPC of 0.144. The total IPC for this configuration is 1.064. Figure 5 (a) shows the performance of both benchmarks as we increase the priority of *h264ref*. We can see that, until prior-

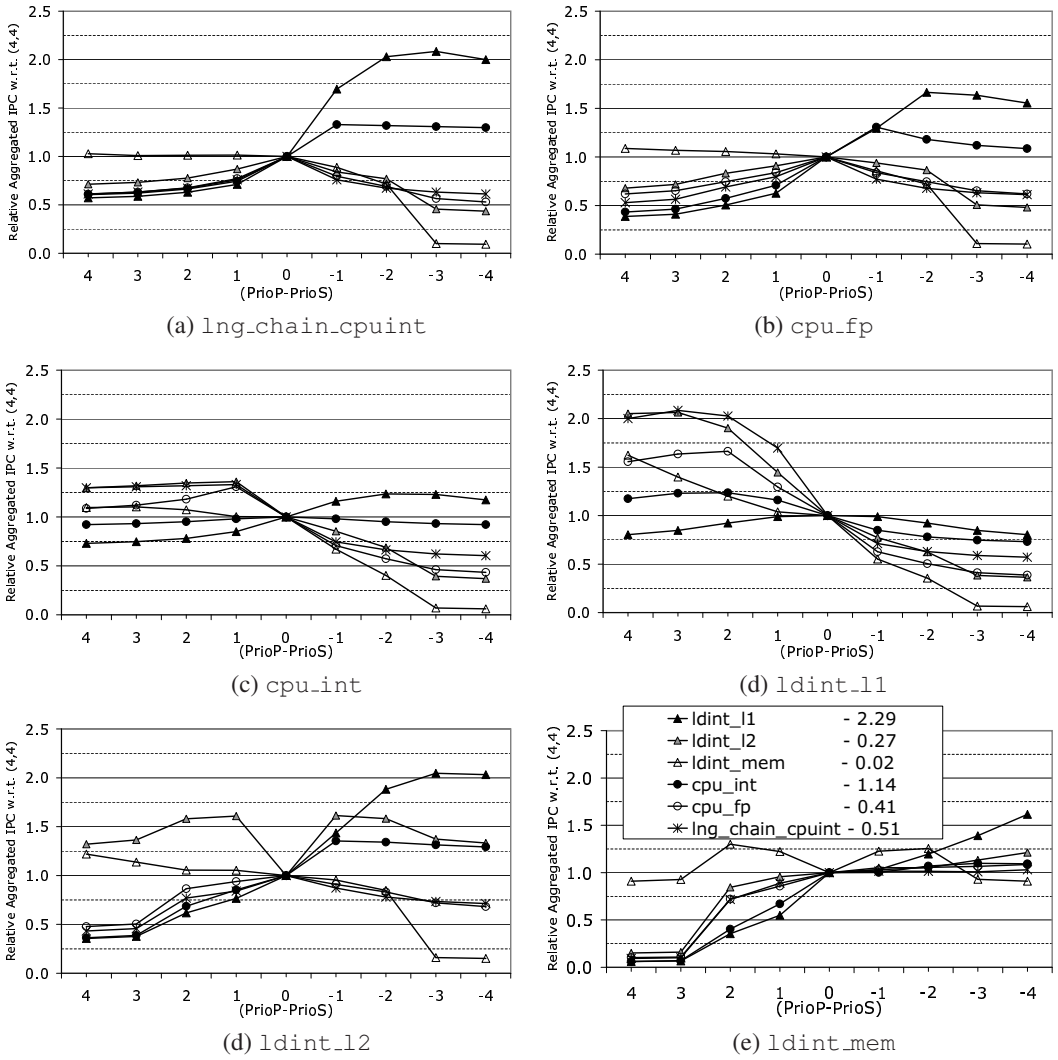


Figure 4. Throughput w.r.t. execution (4,4). The legend shows the single-thread IPC of benchmarks.

ity difference +2, the performance of the *mcf* is reduced by 13.2%, while *h264ref* gains 10.4%. While the gain and the loss are very similar in performance, the overall throughput increases by 7.2%. Further increase in the throughput is possible by degrading low IPC benchmark. The peak IPC is reached when *mcf* runs 32% slower and *h264ref* runs 38% faster than the base case with default priorities. In this case, the overall system performance increases by 23.7%.

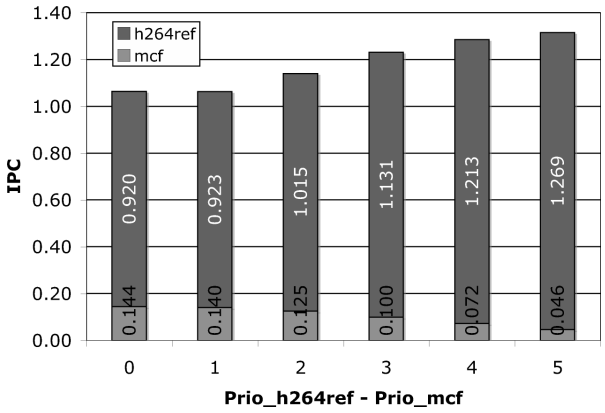
For the second pair, with the default priorities, *applu* has an IPC of 0.500 and completes in 240 seconds. *equake* takes 74 seconds and has an IPC of 0.140. Together, they reach a total IPC of 0.630 (Figure 5 (b)). In this case, the peak combined IPC is obtained when *applu* receives priority +5. It represents a 14% of improvement when compared to the default case.

5.4 Optimizing execution time

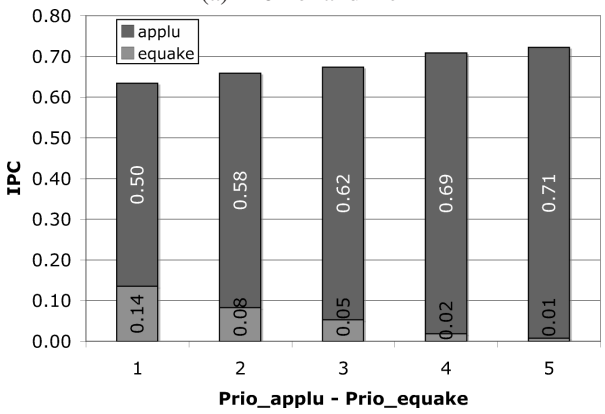
The highest throughput does not always directly translate into the shortest execution time of a whole application [5]. Most of the parallel applications have synchronization points where all the tasks must complete some amount of work in order to continue. Load balancing in parallel application is a hard problem since it is rarely the case where the synchronized tasks finish perfectly at the same time. In other words, usually a task has to wait for other tasks to complete. This could clearly delay the progress of the whole program.

5.4.1 Case Study

In this section we present an example where we are able to improve the overall application execution time by using



(a) h264ref and mcf



(b) applu and equake

Figure 5. Total IPCs with increasing priorities

the prioritization mechanism. In this example, we apply a `LU` matrix decomposition over a set of results produced by a Fast Fourier Transformation (`FFT`) for a given spectral analysis problem. One possible organization of the problem would create a software pipeline where one thread runs the Fast Fourier Transformation, producing the results that will be consumed by the second thread on the next iteration, by applying LU over parts of this output.

Table 4. Execution time, in seconds, of `FFT` and `LU`.

Priority	Priority Difference	`FFT` exec. time	`LU` exec. time	Iteration exec. time
single-thread mode	-	1.86 –	– 0.26	2.12
4,4	0	2.05	0.42	2.05
5,4	+1	2.02	0.48	2.02
6,4	+2	1.91	0.64	1.91
6,3	+3	1.87	2.33	2.33

In our measurement, the `FFT` takes for 1.86 seconds in single-thread mode, and the `LU` takes 0.26 seconds to process its part of the problem. In single-thread mode, the processor would first execute the `FFT` and then the `LU`, thus, each iteration would require 2.12 seconds to complete. In the multi-threaded scenario, there is only `FFT` running in the first iteration to produce the first input of the `LU`. On the remaining iterations, both threads would be running in parallel and the execution time of an iteration would be the execution time of the longest thread. As we can see on the Table 4, when run together in SMT mode, the `FFT` takes 2.05 seconds and the `LU` decomposition takes 0.42 seconds. The `LU` thread would waste 1.63 seconds waiting for the other task to complete. Using the prioritization mechanism, we could increase the priority of `FFT` so it executes faster, reducing the unbalance.

Table 4 shows that the best case consists of running with a priority pair (6,4), which yields an iteration execution time of 1.91 seconds. Effectively this represents a 10% improvement when compared to the execution time in single thread mode (where it would be necessary to run the `FFT` followed by `LU`) and 9.3% of improvement over the default priorities. On the other hand, by applying too much prioritization, it's possible to inverse the unbalance, which normally represents a performance loss (priority (6,3)).

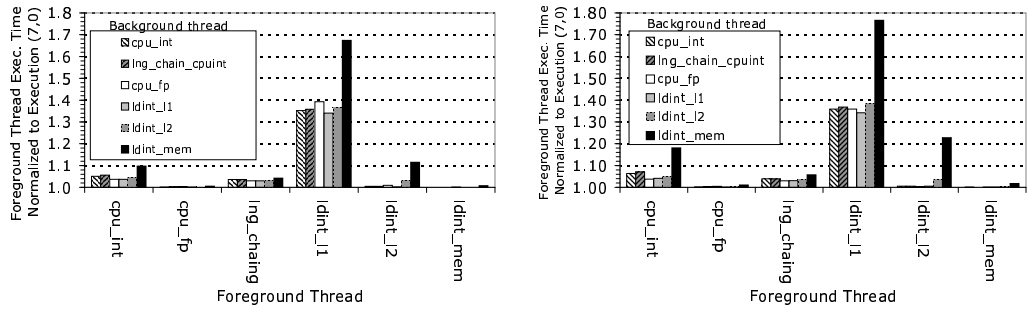
5.5 Transparent execution

Dorai and Yeung [7] propose transparent threads, which is a mechanism that allows background threads to use resources that a foreground thread does not require for running at almost full speed. In POWER5 this is implemented by setting the priority of the "background" thread to 1 [10].

Figure 6 shows the effect of background threads over foreground threads when a foreground thread runs with priority 6 (Figure 6 (a)) and with priority 5 (Figure 6 (b)). We observe that the most effected threads are `ldint_l1`, `cpu_int` and `ldint_l2`, when they are running with a memory-bound background thread.

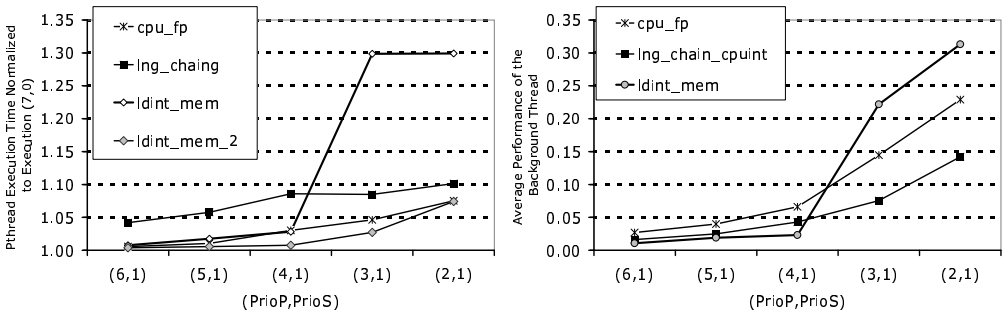
Figure 6(c) presents the maximum effect that a background thread causes on the other foreground threads (`ldint_l2`, `cpu_fp`, and `lng_chain_cpuint`) as we reduce its priority from 6 to 2. In the figure, the different foreground threads run with `ldint_mem` in background as it proved to be the worst case for all combinations.

For `cpu_fp` and `lng_chain_cpuint` the effect of the background thread increases linearly as we reduce the priority from 6 to 2 until about 10% of their ST performance. This is not the case for `ldint_mem` that suffers a sudden increment when its priority is 3 or 2. In the chart, the label 'ld_int_mem_2' represents the performance of the `ldint_mem` when it runs as a foreground thread and the `ldint_mem` is not the background thread. The graph shows that the effect that any other micro-benchmark causes on `ldint_mem` is about 7%. We can conclude that, unless running with another copy of itself, `ldint_mem` can al-



(a) PrioP=6, PrioS=1

(b) PrioP=5, PrioS=1



(c) Worst-case effect of a background thread over the foreground thread

(d) Average IPC of the background thread

Figure 6. Primary thread Execution Time with respect to Single-Thread when *SThread* has priority 1

ways run as foreground thread without much performance degradation.

Finally, Figure 6(d) shows the performance of the background threads. Each point represents the average for all background threads: for example, the point `ldint_mem` (6,1) represents the average performance of the background thread in the experiments (`ldint_mem`, `cpu_int`), (`ldint_mem`, `cpu_fp`), (`ldint_mem`, `lng_chain_cpuint`), (`ldint_mem`, `ldint_l1`), and (`ldint_mem`, `ldint_mem`) using priorities (6,1). We can observe that in the worst performance degradation case (under 10%) for `cpu_fp`, the background threads obtain an IPC of 0.23. For the `lng_chain_cpuint` benchmark this IPC is 0.15.

In general, we can establish that the high-latency threads are the best candidates for foreground thread and the worst background thread. They suffer little impact from a background thread, but heavily effect the performance when running in background. Furthermore, threads with very high performance easily get effected by other threads (see `ldint_l1` on Figure 6 (a)). They may not be suitable to run with a background thread.

6 Conclusions

The IBM POWER5 processor presents two levels of thread prioritization: the first level provides dynamic prioritization through hardware, while the second level is a software-controlled priority mechanism that allows a thread to specify a priority value from 0 to 7. Currently, this mechanism is only used in few cases in the software platforms even if it can provide significant improvements on several metrics. We argue that it is mainly due to the fact that there are no previous works aimed at the characterization of the effects of this mechanism.

In this paper we perform an in-depth evaluation of the effects of the software-controlled prioritization mechanism over a set of synthetic micro-benchmarks, specially designed to stress specific processor characteristics. We present the following conclusions from our micro-benchmarks. First, workloads presenting a large amount of long-latency operations are less influenced by priorities then the ones executing low-latency operations (i.e., integer arithmetic). Second, it is possible, by using the prioritization mechanism, to improve the overall throughput up to two times, in very special cases. However, those extreme improvements often imply drastic reduction of the low IPC thread's performance. On less extreme cases, it is possible to improve the throughput by 40%. And third, we show that, instead of using the full spectrum of priorities, only priorities up to +/-2 should be used, while "extreme" priorities should be used only when the performance of one of the two threads is not important.

In addition, we present three case studies where priori-

ties can be used to improve the total throughput by 23.7%, the total execution time by 9.3% or to have a background thread. Finally, we conclude that the prioritization mechanism in POWER5 is a powerful tool that can be used to improve different metrics. This work opens a path into broader utilization of a software/hardware co-design that allows better balancing of the underlying hardware resources among the threads.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2004-07739-C02-01, TIN-2007-60625, the Framework Programme 6 HiPEAC Network of Excellence (IST-004408) and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. Carlos Boneti is supported by the Catalonian Department of Universities and Information Society (AGAUR) and the European Social Funds.

The authors would like to thank Jordi Caubet, Pradip Bose and Jaime Moreno from IBM, Jose M. Cela, Albert Farres, Harald Servat and German Llort from BSC and Marcelo de Paula Bezerra for their technical support.

References

[1] D. C. Bossen, J. M. Tendler, and K. Reick. POWER4 system design for high reliability. *IEEE Micro*, 22(2), 2002.

[2] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in SMT processors. *In 37th MICRO*, 2004.

[3] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4), 2004.

[4] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transaction on Computers*, 55(7), 2006.

[5] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. On the problem of minimizing workload execution time in SMT processors. In *IC-SAMOS*, 2007.

[6] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. *SIGARCH Computer Architecture News*, 34(2), 2006.

[7] G. K. Dorai and D. Yeung. Transparent threads: Resource sharing in SMT processors for high single-thread performance. In *11th PACT*, 2002.

[8] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *9th HPCA*, 2003.

[9] I. Gelado, J. Cabezas, L. Vilanova, and N. Navarro. The cost of IPC: an architectural analysis. In *WIOSCA*, 2007.

[10] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook, 2005.

[11] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of System Overhead on Parallel Computers. In *4th ISSPIT*, 2004.

[12] H. Q. Le, W. J Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6), 2007.

[13] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997.

[14] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4), 2006.

[15] IBM. PowerPC Architecture book: Book III: PowerPC Operating Environment Architecture. ftp://www6.software.ibm.com/software/developer/library/es-archpub3.zip.

[16] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER5. *Hot Chips*, 2003.

[17] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 Chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2), 2004.

[18] P. M. W. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero. Branch classification for SMT fetch gating. In *6th MTEAC*, 2002.

[19] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. In *IPDPS*, 2003.

[20] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[21] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[22] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *34th MICRO*, 2001.

[23] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd ISCA*, 1995.

[24] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: FAirly MEasuring Multithreaded Architectures. *In 16th PACT*, 2007.

[25] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Measuring the Performance of Multithreaded Processors. In *SPEC Benchmark Workshop*, 2007.