Improving Execution Unit Occupancy on SMT-based Processors through Hardware-aware Thread Scheduling

Achille Peternier, Danilo Ansaloni, Daniele Bonetta, Cesare Pautasso, and Walter Binder[1]

University of Lugano (USI)
Via G. Buffi 13, 6904 Lugano, Switzerland
email: first.last@usi.ch

Abstract

Modern processor architectures are increasingly complex and heterogeneous, often requiring software solutions tailored to the specific hardware characteristics of each processor model. In this article we address this problem by targeting two processors featuring Simultaneous Multi-Threading (SMT) to improve the occupancy of their internal execution units through a sustained stream of instructions coming from more than one thread. We target the AMD Bulldozer and IBM POWER7 processors as case studies for specific hardware-oriented performance optimizations that increase the variety of instructions sent to each core to maximize the occupancy of all its execution units. WorkOver, presented in this article, improves thread scheduling by increasing the performance of floating point-intensive workloads on Linux-based operating systems. WorkOver is a user-space monitoring tool that automatically identifies FPU-intensive threads and schedules them in a more efficient way without requiring any patches or modifications at the kernel level. Our measurements using standard benchmark suites show that speedups of up to 20% can be achieved by simply allowing WorkOver to monitor applications and schedule their threads, without any modification of the workload.

Keywords: multicore, simultaneous multithreading, workload profiling, performance, thread scheduling

[1] corresponding author is Achille Peternier, email: achille.peternier@usi.ch, phone: +41 (0)76 460 38 37

1. Introduction

Since the power wall [1] prevents hardware manufacturers from increasing the processor's clock frequency, modern CPUs embed several cores to increase the computational power through parallelism. Recent trends show that hardware manufacturers are preferring asymmetry and heterogeneity over symmetric and homogeneous designs. Indeed, current state-of-the-art processors have very complex architectures featuring multiple internal components, such as multiple cache levels shared among different cores, Non-Uniform Memory Access (NUMA) [2] controllers and hyperlinks, Simultaneous MultiThreading (SMT) support with several Processing Units (PUs) per core, or ad hoc dedicated units. As a consequence, it is increasingly difficult for software developers to fully exploit the underlying hardware's computational power, as optimal software configurations can vary according to the hardware platform, to the application software architecture, and to the type of workload.

The Operating System (OS) kernel and scheduler try to optimize the performance of applications depending on the available hardware resources. To this end, OS schedulers rely on a limited set of performance indicators (such as the number of cores, CPU time, and memory usage) to drive their optimization strategies. This is often not enough for multithreaded applications running on modern systems, where the complexity and the specific characteristics of the underlying hardware architecture require to use additional information to improve runtime performance through efficient scheduling.

As a case study, in this article we focus on two of these modern architectures and we present a specific, hardware-aware optimization tool based on (1) an automated workload analysis technique relying on a specific set of performance metrics that are currently not used by common OS schedulers, and (2) a hardware-aware optimized scheduler performing scheduling decisions based on hardware resource usage monitoring. Our goal is to use a controller-based approach to characterize the workload of multithreaded and multi-process applications to improve the efficiency of how they share heterogeneous resources.

We focus on two modern micro-architectures that implement very different SMT solutions: the AMD Bulldozer and IBM POWER7 processors. These architectures are good representatives of modern hardware platforms with specific characteristics that cannot easily be exploited by non-hardware-aware approaches. In this context, one of the peculiar characteristics of the Bulldozer architecture is the design of an asymmetric SMT implementation between integer and floating point units, where Floating Point processing Units (FPUs) are shared by two PUs within one same core: two threads may contend for the same FPU units (while integer units are available on a per-PU basis). The IBM POWER7 architecture is based on a more aggressive implementation of SMT, where the instructions coming from up to four threads can be scheduled simultaneously to improve the occupancy of the available execution units on each core. Since each core features two integer and four floating point units, only a proper scheduling of integer- and floating point-intensive threads can take advantage of this improved SMT, otherwise these hardware layouts can have a negative impact on the performance of FPU-intensive workloads.

Our approach is named WorkOver (after Workload Overseer) and corresponds to a Linux daemon that interacts with the OS scheduler to improve thread scheduling of floating point-intensive workloads on SMT processors by taking into account the way hardware execution units are organized into cores and PUs. WorkOver runs in user-space and is based on performance metrics commonly available without any modification of the OS kernel and the monitored applications. Our workload profiling approach relies on hardware performance counters to detect which threads make floating point-intensive computations. Our performance optimization is based on improved thread scheduling by pinning the most FPU-intensive threads to PUs of different cores to reduce contention on shared execution units. In this way, WorkOver provides a transparent bottom-up optimization mechanism, based on (1) automatic workload profiling at runtime through performance counters and (2) hardware-aware dynamic allocation of resources. No further intervention is required, neither to modify the running application (the workload) nor to change the OS scheduler. The tool is a system-wide user-mode daemon collecting information and applying optimization policies on the threads spawned by applications (processes) that have been started with a special command.

This article extends our work presented in [3] by generalizing the approach from a specific CPU model to generic SMT processors and by using two completely different hardware architectures and OSs to validate our generalized approach.

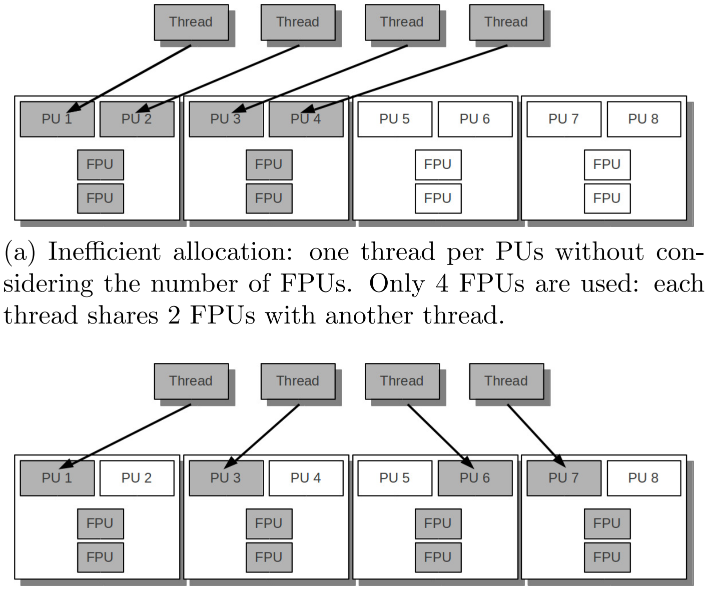
2. Motivation and Approach

Many scientific applications make heavy use of floating point-intensive computations. Consider a scenario in which a multithreaded application performs floating point-intensive computations with variable intensity in all or a subset of its threads. A common OS scheduler would assign FPU-intensive threads to the available SMT units for execution, as it would do for any other application. The scheduler takes metrics such as CPU time consumption into account. However, prevailing schedulers included in most OS distributions do not consider the way the executed workload is using the hardware resources.

On modern architectures, it makes a significant difference to schedule threads by taking into account the characteristics of the underlying hardware. For simplicity, let's assume that a multithreaded application with 8 running threads has a subset of 4 threads performing FPU-intensive operations. The execution of such application on an AMD Bulldozer four-core processor with with 2 PUs on each core (thus seen as a processor with 8 PUs in total) could potentially result in an inefficient use of computing resources. If the OS scheduler scatters the 4 floating point threads to 4 PUs used by two cores (see Figure 1a), the total number of FPUs used will be 50% less than when the same 4 threads are scheduled one per core (Figure 1b).

This scenario can be even more detrimental to performance when it happens on a IBM POWER7 four-core processor with 4 PUs on each core: in such a case (see Figure 2a), only the FPUs of a single core would be used by the four threads, while FPUs belonging to other cores would idle unless a better scheduling (such as the one depicted in Figure 2b) is applied.

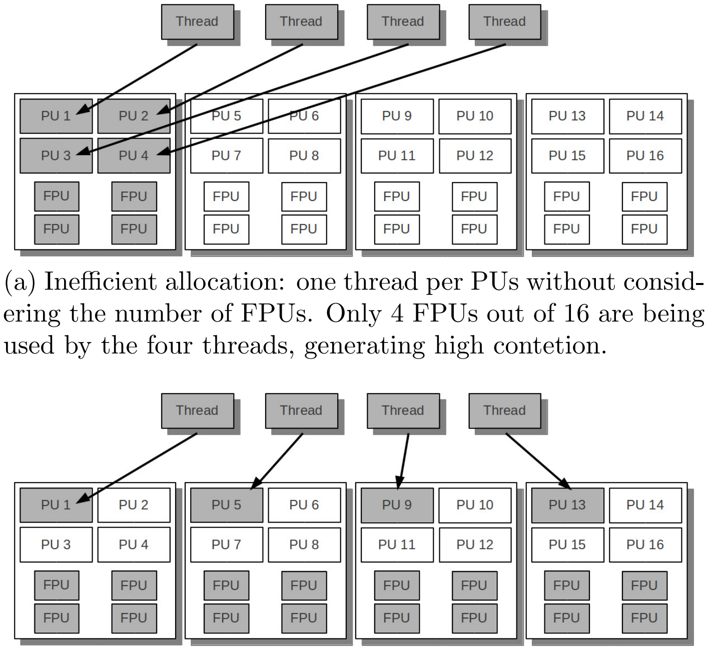
Our goal is to prevent such inefficient scheduling by identifying and binding threads performing floating point computations to specific PUs. We rely on hardware performance counters to measure the number of FPU-related operations performed by each thread. Based on this information we identify at runtime which threads would benefit from our approach, in order to have an interleaved distribution of FPU- and non-FPU-intensive threads over the available PUs. Thanks to the regular monitoring, our solution is able to change thread scheduling on-the-fly, according to the previous measurement time lapse. In this way, threads that change their integer/FPU usage intensity over time are rescheduled in a more efficient way, e.g., when a thread pool reuses the same threads for doing different tasks or when one thread's FPU usage varies.



(a) Inefficient allocation: one thread per PUs without considering the number of FPUs. Only 4 FPUs are used: each thread shares 2 FPUs with another thread.

(b) Optimal allocation: one FPU-intensive thread per core. All the 8 available FPUs are used: each thread uses 2 dedicated FPUs.

Figure 1: Inefficient vs optimal scattering of 4 floating point-intensive threads on a AMD Bulldozer processor.



(a) Inefficient allocation: one thread per PUs without considering the number of FPUs. Only 4 FPUs out of 16 are being used by the four threads, generating high contetion.

(b) Optimal allocation: one thread per core. 16 FPUs used: each thread uses 4 dedicated FPUs.

Figure 2: Inefficient vs optimal scattering of 4 floating point-intensive threads on a IBM POWER7 processor.

Our dynamic approach does not require any code analysis, annotation, or offline simulation. Since performance counters are a limited hardware resource (only a small number of them can be used at the same time), they are usually not used by OS schedulers to perform continuous, system-wide monitoring of all the running threads because user-space applications would not be able to use additional counters for their own needs. For example, if an OS scheduler would reserve two performance counters out of the maximum of six that can be simultaneously activated (both on the AMD Bulldozer and IBM POWER7 processors) to monitor all the running threads for its needs, only four counters would be left for usage by other applications measuring different events on the same threads. Performance counters are a precious and limited resource, since the maximum number of simultaneously usable counters is much lower than the total number of available performance events (e.g., 82 on AMD Bulldozer and 537 on IBM POWER7).

Our approach, based on a centralized monitoring daemon that applies optimization policies only to threads spawned within user-specified processes, is executed at the user-level, without interfering with the rest of the system. The OS scheduler only receives hints about where to schedule specific threads. All the threads started within selected applications are regularly monitored to observe whether they are generating FPU-related workload or not, so that improved FPU-occupancy scheduling can change and be refined at runtime. According to the number of floating point operations executed by each thread, WorkOver classifies the most FPU-intensive threads (up to maximum number automatically determined in function of the available FPUs, cores and the way they are distributed) and binds them to PUs by interleaving the floating point intensive ones with the integer (or less floating point intensive) ones, instead of randomly distributing them across all PUs (as default schedulers normally do). Whereas other scheduling policies are based on other well researched hardware resources (e.g., NUMA memory layout and shared caches [4, 5, 6]) and target other purposes (e.g., power efficiency [7, 8, 9, 10]), in this work we focus on a different shared hardware resource (floating point processing units) to improve performance.

3. Background

3.1. AMD Bulldozer Processors

This section describes in detail the internal architecture of the AMD Bulldozer processor family[2].

The processor embeds multiple cores (referred to as *processing modules* in the AMD documentation). Each core features a front-end to fetch and decode instructions, caches (a larger L3 cache is shared by all the modules being part of the same CPU), a branch prediction unit, out-of-order instruction schedulers, and integer and floating point processing pipelines. Each core has 2 PUs (referred to as *cores* in the AMD documentation) and can run two threads simultaneously. However, unlike IBM's and Intel's SMT implementations (where two or more threads share all the resources of the core), AMD integrated separate integer pipelines with their own scheduler and retire units.

As a consequence, a Bulldozer has two Arithmetic and Logic Units (ALUs) per PU, thus one thread can perform a maximum of two integer operations in a single cycle at the hardware level. FPUs are also present through two dedicated units, but they are instantiated per core and not per PU (see Figure 3). Two threads performing floating point operations scheduled on the two PUs of the same core will share 2 FPUs, while the same threads scheduled on PUs belonging to two different cores will use a total of 4 FPUs (2 FPUs for each thread exclusively).

The result is that Bulldozer PUs behave more like a dual-core system for integer-intensive tasks and more like a single-core with SMT for floating point computations (including SIMD operations provided by MMX/SSE and similar extensions[3]).

Each core is also completed by an L2 cache shared across its two PUs (while L1 are instantiated per PU and a larger L3 is shared by all the PUs/cores of one same processor).

[2] `http://www.amd.com/us/products/desktop/processors/amdfx/Pages/amdfx.aspx`

[3] The 2 FPUs can also be used together to perform the new 256-bit operations introduced through the Advanced Vector Extensions (AVX).

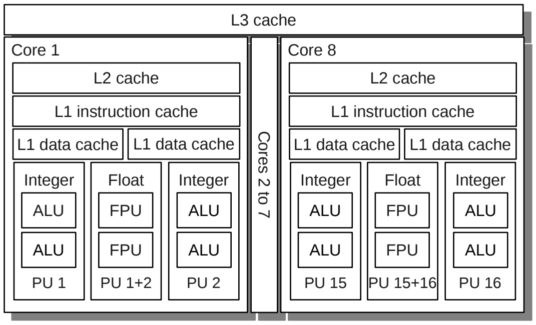


Figure 3: Architecture of a AMD Bulldozer processor with 8 cores and 16 PUs.

3.2. IBM POWER7 Processors

The POWER7 processor (see Figure 4) is composed of several cores and is mainly designed on the principle of out-of-order instruction execution to improve the occupancy of the available internal units. Each core features twelve execution units as 2 FiXed point Units (FXUs), 2 load store units, 4 double precision floating point units, 1 vector unit, 1 branch unit, 1 condition register unit, and 1 decimal floating point unit. In order to feed all the available execution units with a sufficient stream of instructions, POWER7 processors introduced a new SMT mode with 4 PUs capable of executing up to four instruction threads simultaneously on a single core. POWER7 processors can be configured to run in SMT1 (with 1 PU per core), SMT2 (2 PUs per core), and SMT4 (4 PUs per core). The SMT4 mode improves the POWER7 processor's throughput by increasing the execution-unit occupancy efficiency of each core [11, 12].

The POWER7 processor architecture of each core is completed by two L1 (instruction and data) caches of 32 kB each, 256 kB of L2 cache, and 4 MB of L3 cache. L3 cache can be shared among different cores of one same processor according to automatically detected access patterns: a specific core's L3 cache region can be accessed by other cores by paying a penalty in the latency.

3.3. Hardware Performance Counters

Performance counters (or performance events) are registers embedded into processors to keep track of hardware-related events such as cache misses, number of CPU cycles, retired instructions, etc. Counters are vendor and

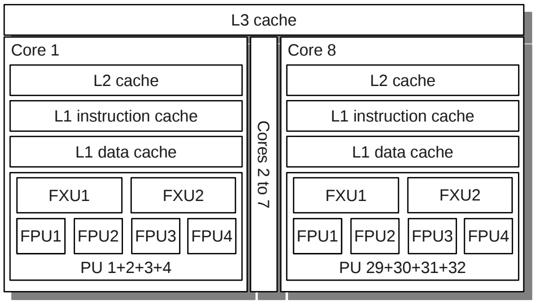


Figure 4: Architecture of a IBM POWER7 processor with 8 cores and 32 PUs.

hardware-architecture specific. Selected counters can be used for monitoring events either on specific cores (per-core profiling) or on specific threads (per-thread profiling). Since performance counters are directly implemented at the processor level, their overhead is very low. Counters are commonly used as efficient instruments for empirical profiling [13], providing insight on what is happening at the hardware level.

In our case, we use counters for workload profiling to dynamically determine which threads (and in which percentage) execute floating point instructions. Based on this information, we schedule them on specific PUs to balance the occupancy of the execution units available on the two target hardware architectures.

4. WorkOver Design

4.1. Requirements and Dependencies

To verify and translate our ideas into practice, we extended our Java low-level monitoring library[4] (described in [14]) into WorkOver, a tool for Linux-based systems written in C++. WorkOver uses `hwloc`[5] for inspecting the underlying hardware configuration and for dynamically extracting information on its topology to enable automatic hardware-awareness. Additional information not provided by `hwloc` about the number and the way execution units are distributed among cores and PUs is read from processor-architecture

[4] `http://sosoa.inf.unisi.ch/overseer`

[5] `http://www.open-mpi.org/projects/hwloc/`

specific configuration files that are loaded according to the detected processor family. These configuration files also include the performance counter strings relative to each architecture to improve portability of WorkOver to other platforms. Hardware performance counters are managed through `libpfm`[6], which is a light-weight library for gathering performance events. We used `libmonitor`[7] for intercepting the creation and termination of threads and processes. All the multithreading mechanisms supported by `libmonitor` are then supported by WorkOver, which comprise pthreads, signals, dlopen, OpenMP and MPI.

4.2. Software Architecture

WorkOver is designed with a client-server architecture (see Figure 5). The server is a centralized daemon that receives the list of process and thread IDs to monitor and optimize. The overhead introduced by the monitoring infrastructure and the daemon is very low (below 2%). In Figure 5, for instance, there are three applications being monitored. These applications are started just like any other Linux process through the shell prompt by preceding the application name with the *workover* command (e.g., `shell:/$ workover executable`). This uses `libmonitor` to register callbacks that are invoked each time a new process or thread is created and terminated by the application. To do so, `libmonitor` overrides functions such as `main`, `pthread_create`, `fork`, `dlopen`, `MPI_Init`, OS signals, etc. (see [15]). The advantage is that applications built using the same multithreading mechanisms supported by `libmonitor` can be monitored without any change nor recompilation. Therefore, WorkOver is totally transparent to the application being monitored through its Daemon Client.

When a new thread is created, the Daemon Client notifies the server about the new Process IDentifiers (PIDs) to be tracked. In this way, the WorkOver server updates an internal list containing all the PIDs related to each monitored application. By knowing a thread's PID, the server assigns and monitors specific hardware performance counters for that thread instance. This allows to obtain a fine-grain monitoring of the hardware performance events generated by each process (and by each thread spawned within each process, in the case of multithreaded applications).

[6] `http://perfmon2.sourceforge.net/`

[7] `https://outreach.scidac.gov/projects/libmonitor/`

The performance events are used to sort the list of the monitored threads to identify which threads are putting more stress on the FPUs among the most active ones (identified by the number of CPU cycles they consumed during the previous sampling period). The control loop is closed by setting the CPU-thread affinity mask of the top FPU-intensive threads to bind them to the available PUs in a way that:

1. prevents excessive stress on some core by making sure that top FPU-consumers are scheduled on PUs belonging to different cores;

2. avoids a single category of threads (such as integer- or FPU-consumers only) from running on one same core, thus not using all the execution units available;

3. reduces thread migrations through a caching mechanism that schedules the same thread to the same PU as long as possible (thus reducing cache refilling due to unnecessary PU changes). WorkOver keeps a cumulative PU usage history for each PID received: when the top-consumers list is updated, threads with a PID that was already part of this group are re-scheduled to the same PU.

Non FPU- and less FPU-intensive threads that are not among the top-consumers are scheduled by the OS scheduler and are left untouched by WorkOver.

4.3. Considered Performance Events

The hardware performance events monitored by WorkOver to perform the workload profiling depend on the underlying hardware architecture and are thus specific to either the AMD Bulldozer or the IBM POWER7 processor.

On the AMD Bulldozer we used:

- **PERF_COUNT_HW_CPU_CYCLES** to measure the total number of CPU cycles consumed by a thread. The reported value is incremented only during the thread execution time and is not affected by thread migration. This counter measures all the CPU cycles executed, without categorizing them as being spent doing integer or floating point tasks.

- **CYCLES_FPU_EMPTY** keeps track of the number of CPU cycles the floating point operation scheduler is empty. Since the floating point scheduler also stores FP-instructions using extended functionalities (such as MMX/SSE/AVX), this counter works independently of the extension set being used[8].

On the IBM POWER7 we considered:

- **PM_CYC**, which is the IBM version of the PERF_COUNT_HW_CPU_CYCLES counter previously described. Its semantics is identical.
- **PM_FLOP** counts the number of floating point operations that have been generated during a thread's execution time.

Our assumption (verified hereafter in Section 5.2) is that these counters give an empirical estimate of the floating point load that each thread is generating. By polling these values regularly (i.e., every second[9] in our experiments based on scientific workloads), WorkOver can dynamically identify which threads have recently been executing floating point-intensive workloads. We assume that a thread showing an intensive FPU-related activity during the monitoring sampling phase will probably continue during the next iteration, and thus will be scheduled in a different way against threads not using the FPU units. Since we regularly update the workload profiling, when a thread changes its behavior, we can dynamically change its scheduling as well. This option, for example, is not permitted by other approaches relying on *a priori* thread pinning.

FPUs are not the only hardware resource shared by two or more PUs. In particular, shared L2 caches available in both the AMD Bulldozer and IBM POWER7 processor cores can have critical effects on performance. To differentiate the speedup contribution given by improved FPU usage from possible effects due to the L2 cache, we use two additional performance counters

[8] This includes 256-bit AVX instructions, where both FPUs are coupled to perform operations. Since FPUs are used *after* being scheduled by the floating point scheduler, threads abundantly using 256-bit AVX instructions will tend to fill the FP-scheduler more often and will be identified as intensive FPU consumers.

[9] Since our evaluation is based on scientific workloads requiring up to several minutes to complete, we found that 1 sec is a reasonable monitoring sampling frequency. Since the total overhead of WorkOver is very low, an higher sampling frequency would only reduce the accuracy of the optimizations without any payback. On the other hand, short-living and fast-changing threads would benefit from a faster refresh rate. The sampling frequency interval is also part of the WorkOver configuration files and can be customized at will according to the context.

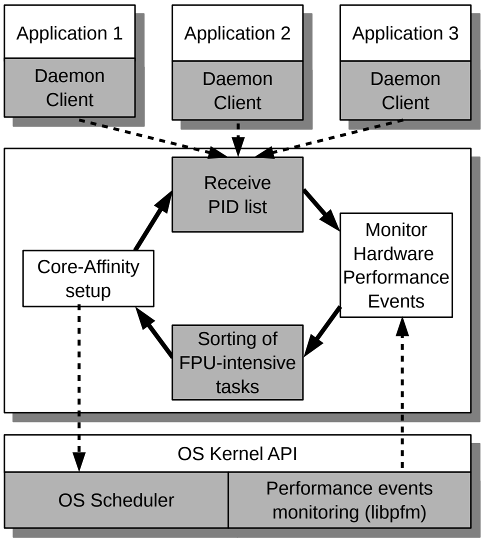


Figure 5: Adaptive control loop implemented by the WorkOver daemon.

(L2_CACHE_MISSES on AMD Bulldozer and PM_L2_LDST_MISS on IBM POWER7) to measure the number of cache misses. We only use this information during our evaluation to observe the impact of the WorkOver scheduling at that level: WorkOver does not use these counters. When a significant speedup is detected with almost no variation in the total number of cache misses, it is reasonable to suppose that such a speedup is not introduced by an accidentally improved cache efficiency produced by WorkOver.

5. Evaluation

5.1. Testing Environment

Experiments are performed on two different machines. The first one (henceforth referred to as AMD-Bull) is a 4 CPU Dell PowerEdge M915 with 128 GB of RAM. Each CPU is an AMD 6282SE 2.6 GHz processor with 8 cores including 2 PUs each, for a total of 16 PUs[10]. This machine features 8 NUMA nodes with 2 nodes per CPU[11]. To avoid latencies introduced by the

[10] Using the AMD terminology, this corresponds to a CPU with 8 modules and 16 cores.

[11] The AMD Bulldozer architecture extends NUMA inside sockets, splitting each CPU into two additional NUMA nodes with intra-CPU latencies lower than extra-CPU ones.

non-uniform architecture, and since our work is not aiming at NUMA-aware scheduling (a topic already well covered by research), all the experiments have been performed by using a single NUMA node and local memory (that is, using the RAM directly connected to that CPU NUMA node, without accessing the InterConnect). Under these settings, the machine corresponds to a single CPU server with 4 cores/8 PUs (as the one depicted in Figure 1) and 16 GB of RAM. As a consequence, all the cores share the same L3 cache.

The second machine (henceforth referred to as IBM-P7) is an IBM 4 CPU P755 with 128 GB of RAM. Each CPU is an IBM POWER7 3.3 GHz processor with 8 cores. Each core can activate and use up to 4 PUs (the activation is statically done by means of a command-line tool). This machine features 4 NUMA nodes (one per CPU). Similarly to AMD-Bull, all the experiments have been performed by using a single NUMA node and local memory. Under these settings, IBM-P7 corresponds to a single CPU server with 8 cores and a maximum total of 32 PUs. In this case, too, all the cores share the same L3 cache.

Where available, energy saving, dynamic CPU frequency scaling, and Turbo mode have been disabled for improved experiment repeatability.

The OS used is Ubuntu Linux Server 64bit version 11.10 on AMD-Bull and RedHat Enterprise Linux 64bit for PowerPC version 6.3 on IBM-P7. C++ code is compiled using GCC version 4.6.1 on AMD-Bull and version 4.7.3 (via Advance Toolchain 6.0-1) on IBM-P7. We used Java through the Oracle JDK 1.7.0_2 Hotspot Server VM on AMD-Bull and IBM J9 1.7.0 SR3 on IBM-P7.

Our evaluation approach is on realistic case studies and relies on two established benchmark suites: Spec.CPU[12] and SciMark2.0[13]. The Spec.CPU suite perfectly fits our needs since its benchmarks are organized into two main categories: integer and floating point. For our evaluation, we used seven randomly chosen benchmarks of each group. SciMark2.0 performs a set of numeric intensive computations: Fast-Fourier Transformation (FFT), Jacobi Successive Over-Relaxation (SOR), Monte Carlo integration (MC), sparse Matrix Multiply (MM), and dense LU matrix factorization (LU), each with different levels of stress on the FPUs.

[12] http://www.spec.org/

[13] http://math.nist.gov/scimark2/

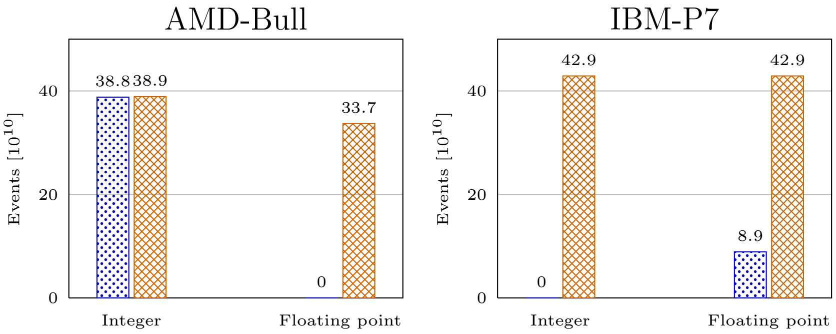


Figure 6: Performance counters microbenchmark comparing empty FPU cycles on AMD-Bull (░) and total FPU operations on IBM-P7 (░) with the total number of CPU cycles (▩).

5.2. Workload Profiling

To validate the usage of performance counters as an instrument for workload profiling and to verify the accuracy they provide in the identification of floating point intensive code, we first performed two synthetic experiments running two workloads: *Integer* and *Floating point*.

For the first experiment, a series of 4x4 matrix multiplications (a common operation used in computer graphics) is executed using only integer or floating point variables. The execution of the workloads is monitored through the counters previously described. Results are reported in Figure 6.

On AMD-Bull, the CYCLES_FPU_EMPTY counter is effectively helping in determining the number of FPU-related operations. The *Integer* configuration, as expected, is confirmed as not consuming FPU operations (that is, the FPU pipeline is empty for almost all the CPU cycles consumed by the application). By using the *Floating point* configuration, where only floating point operations are executed, we observe how the number of CYCLES_FPU_EMPTY events is close to zero. As expected, the FPU pipeline is constantly filled with new operations, since the code contains mainly FPU-related ones.

On IBM-P7, the PM_FLOP counter reports a number close to 0 for the *Integer* test as expected, since no floating point operations are involved. The *Floating point* test shows that several millions of floating point operations have been executed during the experiment.

If we compare the ░ bars between AMD-Bull and IBM-P7 it may surprise how clearer are the results obtained on AMD-Bull than IBM-P7. In fact, while on AMD-Bull we are comparing two counters measuring cycles, on IBM-P7 we compare cycles against (floating point) instructions. Since an instruction can take several cycles to complete, the gap between the counters is higher on IBM-P7. In addition, the PM_FLOP counter is "direct", that is, it is keeping track of the total number of floating point operations, while the CYCLES_FPU_EMPTY counter tells us "indirectly" how much stress on the FPUs a thread is generating by giving an insight on how often the FPU pipeline is idle (empty). For this reason, and for the sake of clarity, it is important to remember that on AMD-Bull a higher number of CYCLES_FPU_EMPTY events means low consumption of FPU operations, while on IBM-P7 is the contrary: a higher count of PM_FLOP events means intensive FPU usage. Finally, since performance counters are hardware-specific, they should not be used for inter-architectural comparisons. In our case, performance counters are used on each target architecture to locally profile the workload and to provide some values that we can use to identify a specific behavior, but these values make sense only within each machine and cannot be merged with results obtained on a different one.

Nevertheless, the most important information provided by this experiment is the ratio between the counters used to measure the presence (or absence) or FPU operations and the total number of cycles, which provides a value that we can use to measure the impact that threads have on FPUs and use this information as input to improve scheduling via WorkOver.

To confirm these results, we applied the same methodology to profile the FPU workload generated by Spec.CPU and SciMark2.0 benchmarks. Results for AMD-Bull are reported in Figure 7 and Figure 8, while results for IBM-P7 are depicted in Figure 9 and Figure 10. We use these values to determine a ratio given by Equation 1 on AMD-Bull and by Equation 2 on IBM-P7.

$$FPUusage = 1 - \frac{EmptyFpuCycles}{TotCpuCycles} \tag{1}$$

$$FPUusage = \frac{TotFpuOps}{TotCpuCycles} \tag{2}$$

We use *FPU usage* to characterize the FPU workload generated by each thread. Spec.CPU and SciMark2.0 FPU usages are shown in Table 1 (AMD-Bull) and 2 (IBM-P7).

Since on IBM-P7 we are dividing FPU operations by CPU cycles (as discussed before), this ratio is significantly lower than the one obtained on

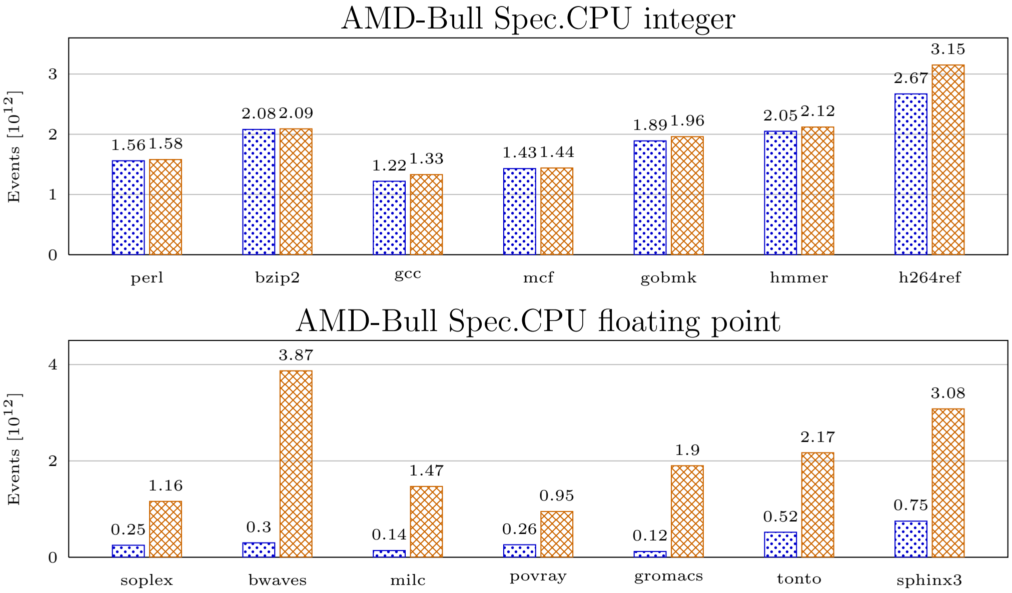


Figure 7: Empty FPU cycles (░) and CPU cycles (▩) count for each test of the Spec.CPU benchmark executed on AMD-Bull.

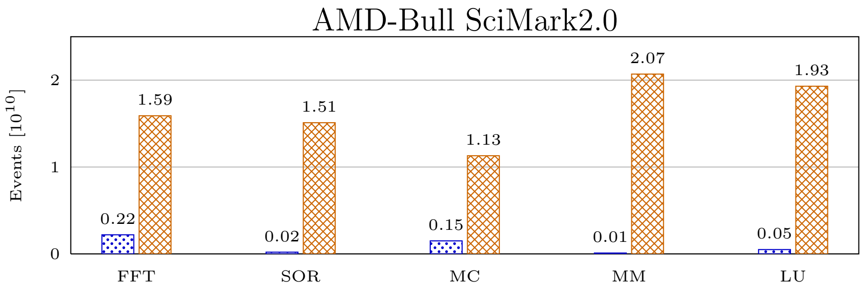


Figure 8: Empty FPU cycles (░) and CPU cycles (▩) count for each test of the SciMark2.0 benchmark executed on AMD-Bull.

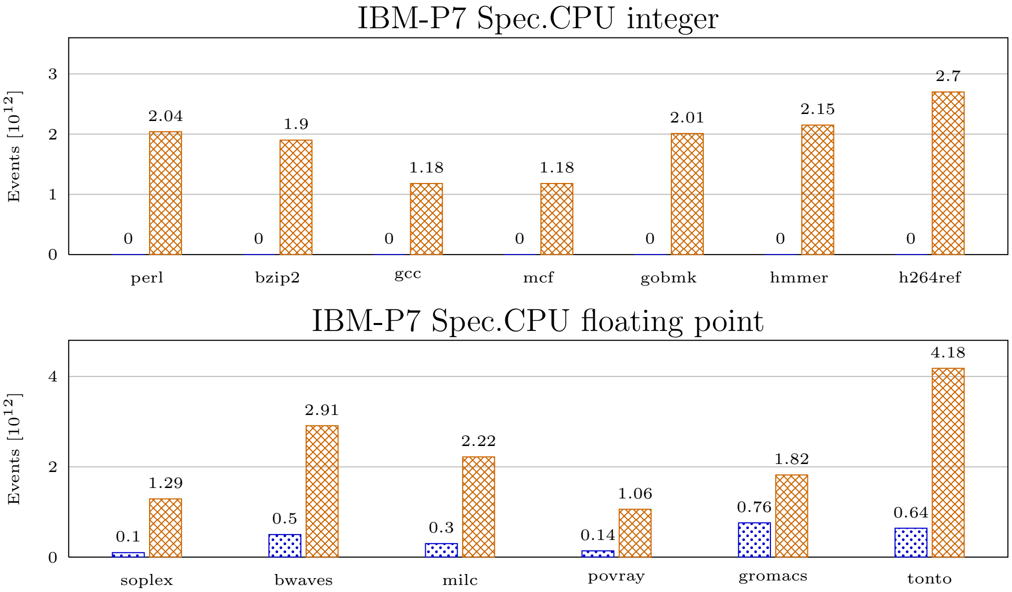


Figure 9: FP operations (▦) and CPU cycles (▩) count for each test of the Spec.CPU benchmark executed on IBM-P7.

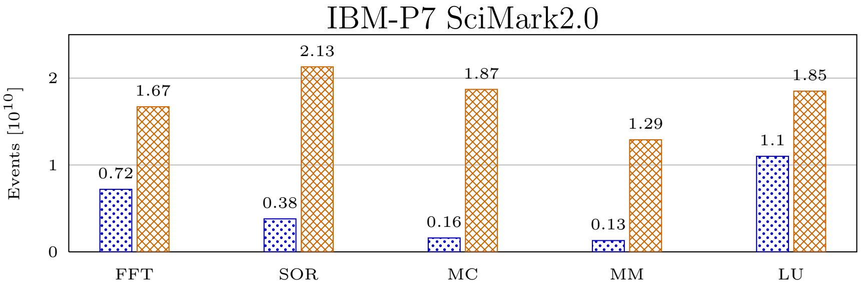


Figure 10: Empty FPU cycles (▦) and CPU cycles (▩) count for each test of the SciMark2.0 benchmark executed on IBM-P7.

AMD-Bull FPU usage − *Spec.CPU integer*						
perl	bzip2	gcc	mcf	gobmk	hmmer	h264ref
0.01	<0.01	0.08	<0.01	0.03	0.03	0.15
L2 cache miss rate						
0.12	0.5	0.84	1.62	0.08	0.07	0.07

AMD-Bull FPU usage − *Spec.CPU floating point*						
soplex	bwaves	milc	povray	gromacs	tonto	sphinx3
0.79	0.92	0.91	0.72	0.94	0.76	0.75
L2 cache miss rate						
2.14	0.98	1.74	0.01	0.05	0.24	1.63

AMD-Bull FPU usage − *SciMark2.0*				
FFT	SOR	MC	MM	LU
0.86	0.99	0.87	0.99	0.97

Table 1: AMD-Bull FPU usage for the Spec.CPU and SciMark2.0 benchmarks as given by Equation 1. Spec.CPU benchmark characterization is completed by L2 cache miss rates.

AMD-Bull (where we divide idle cycles by total cycles). However, since these values are only an internal indicator used to differentiate threads, absolute values do not matter and they should not be used, e.g., to perform a comparison on how efficient is the AMD Bulldozer or IBM POWER7 processor in doing floating point maths.

Thanks to the *FPU usage* metric, and by regularly updating the values reported by performance counters, WorkOver monitors which threads are performing more FPU-intensive computations and forces the scheduler to assign them to a core where no other FPU-intensive threads are already running. This mechanism is at the basis of the controller used by WorkOver, which is evaluated in the next experiment.

The two tables are completed by an additional index: the L2 cache miss rate. This rate is defined by the number of L2 cache misses divided by the number of CPU cycles. We use the frequency of L2 cache misses to have a raw approximation of the level of contention that each benchmark puts on the L2 cache. Since each core shares its FPUs and its L2 cache among the available PUs, later on we use this rate to differentiate the performance contribution given by improved FPU usage and from better cache locality.

SciMark2.0 benchmarks are very compact in terms of both memory consumption and code length, efficiently fitting within CPU caches. As a conse-

IBM-P7 FPU usage − *Spec.CPU integer*						
perl	bzip2	gcc	mcf	gobmk	hmmer	h264ref
<0.001	<0.001	<0.001	0.002	0.004	<0.001	<0.001
L2 cache miss rate						
0.004	0.017	0.030	0.045	0.003	0.004	0.003

IBM-P7 FPU usage − *Spec.CPU floating point*						
soplex	bwaves	milc	povray	gromacs	tonto	sphinx3
0.074	0.18	0.13	0.13	0.42	0.15	-
L2 cache miss rate						
0.027	0.014	0.018	<0.001	0.003	0.004	-

IBM-P7 FPU usage − *SciMark2.0*				
FFT	SOR	MC	MM	LU
0.43	0.18	0.09	0.11	0.61

Table 2: IBM-P7 FPU usage for the Spec.CPU and SciMark2.0 benchmarks as given by Equation 2. Spec.CPU benchmark characterization is completed by L2 cache miss rates.

quence, they generate very few cache misses (several orders of magnitude less than Spec.CPU). For this reason, we can ignore this aspect when running SciMark2.0 on our testing hardware.

The Spec.CPU sphinx3 benchmark on IBM-P7 gives a compilation error and we have not been able to run it on this machine.

5.3. Case Study

In this section, we evaluate the speedup achievable with WorkOver when the system executes heavy numeric computations. When only a few threads are active, the OS is likely to schedule them on different cores. However, when the system utilization increases, threads executing floating point-intensive workloads may be scheduled on cores that are part of the same core. When used, WorkOver monitors the FPU usage ratio of all running threads spawning within user-selected applications. Every second, the most floating point intensive threads are bound to PUs belonging to different cores, decreasing the contention on shared FPUs and giving room to the OS scheduler to schedule less FPU-intensive or integer-only threads on the remaining PUs.

On AMD-Bull, using 8 PUs over 4 cores as configured for our testing environment, this implies that the scheduling of the top 4 FPU-intensive threads is restricted to the first PU of each core. The following 4 threads are scheduled on the second PU of each core, while remaining threads have no restrictions. To reduce unnecessary thread migrations, the caching mechanism previously described prevents threads staying within the first or second group of 4 threads for more than a sampling cycle from being rescheduled on a different core.

On IBM-P7, we perform experiments using two different SMT levels: SMT2 and SMT4. When configured in SMT2, IBM-P7 uses 16 PUs and 8 cores, while with SMT4 it uses 32 PUs and 8 cores.

5.4. Multithreaded Spec.CPU on AMD-Bull

In this experiment we measure the wall-time required to run 4 instances of Spec.CPU integer and 4 instances of Spec.CPU floating point benchmarks on AMD-Bull. The idea is to automatically let WorkOver analyze and decide how to schedule their threads to improve the usage of available execution units. After testing many pairs of benchmarks, we include and discuss here the results obtained with two pairs (*hmmer+povray* and *mcf+sphinx3*) that are particularly representative of two recurrent behaviours that we observed (in all cases, including other pairs, we have been able to identify the presence of one or both of these behaviors with different weights).

We run the experiments using three different configurations: (1) an intentionally suboptimal scheduling (░░) aggregating similar workloads to the PUs of one same core (that is: integer with integer, floating point with floating point); (2) the optimized scheduling (▩▩) putting heterogeneous threads together (an integer thread with a floating point thread); and (3) by not using WorkOver at all and letting the default OS scheduler run (▨▨). Results are reported in the left chart of Figure 11.

Since one AMD Bulldozer core shares the FPUs and the L2 cache among its two PUs, we also report (in the right chart of the same Figure) the number of L2 cache misses generated during the experiment. This counter gives us an additional information about which hardware resource (FPU or L2 cache) is more responsible for the performance gain or degradation.

Results. Each entry is computed as the mean value after ten independent runs. Results are very stable for the intentionally suboptimal baseline and WorkOver optimized one (with a standard deviation below 1%). Things are different concerning the default OS scheduling: since the scheduler cannot characterize threads as integer- or FPU-intensive, it considers them all the same and runs them on the available PUs. To improve locality and prevent performance degradation due to unnecessary migrations, it tends to keep one thread tied to a specific PU until the overall workload changes. In this way,

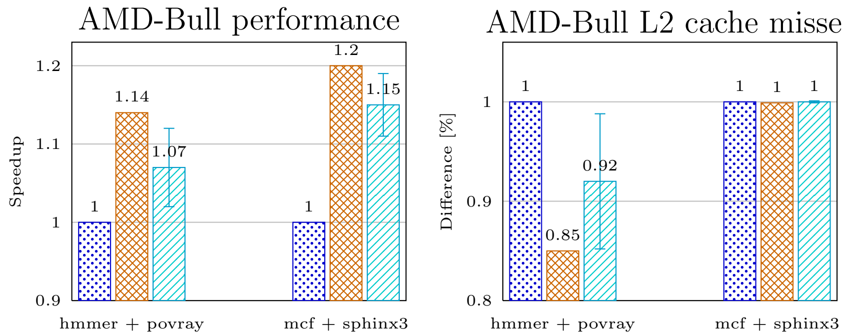


Figure 11: Spec.CPU speedups (top) and cache miss differences (bottom) on AMD-Bull, using an intentionally inefficient baseline (▨▨) versus using WorkOver to improve performance (▨▨) or letting the default OS scheduler do the job (▨▨).

in some cases the scheduler manages to scatter integer and floating point threads in an optimal way, while in other cases it assigns them more like our intentionally suboptimal configuration.

Our optimal and suboptimal configurations somehow emulate the range of possible performance that the scheduler can (non-deterministically) achieve, while by using WorkOver the optimal deployment is immediately identified and maintained (as it could be achievable by manually pinning threads to the proper PUs *a priori*).

Results show that WorkOver is from 15% to 20% faster than the suboptimal baseline and by 6% to 10% faster (on average) than the default scheduler. WorkOver also provides stable results among independent runs of the same benchmark, while the default scheduler is more noisy.

At this point, we have shown that WorkOver is concretely bringing some advantages to the system, but how can we be sure that this speedup is really coming from improved execution unit occupancy and not from a better cache usage? According to our measurements, speedups are supported by both these events but with a different weight for each pair. The two pairs we selected are well representative of this dual-contribution: if we compare the speedup and difference in cache misses of the *hmmer+povray* pair, we can easily observe that they are strictly correlated (a +14% speedup is followed by a similar percent of reduction in the number of L2 cache misses). The cache miss rate of Table 1 gives us some insight: *hmmer* has a rather low value (0.07) and *povray* even less (close to 0). This means that these two benchmarks are stressing the L2 cache in a significantly different way. Now, since on each AMD-Bull core the L2 cache is shared by its two PUs, when we put two *hmmer* threads on the same core, there is more contention on the L2 than when a *hmmer* thread is executed in combination with a less cache-intensive thread such as *povray*.

The *mcf+sphinx3* case is different. Here the cache miss difference among the three configurations is below 0.1% but we measure speedups of up to 20%. In fact, these two benchmarks have a similar cache miss rate of 1.62 and 1.63 respectively, thus are putting contention on the L2 cache in a similar way. In this case, it is difficult to justify the speedup by such a small delta in the cache miss count. The performance gain is more likely introduced by a better workload distribution over the available execution units.

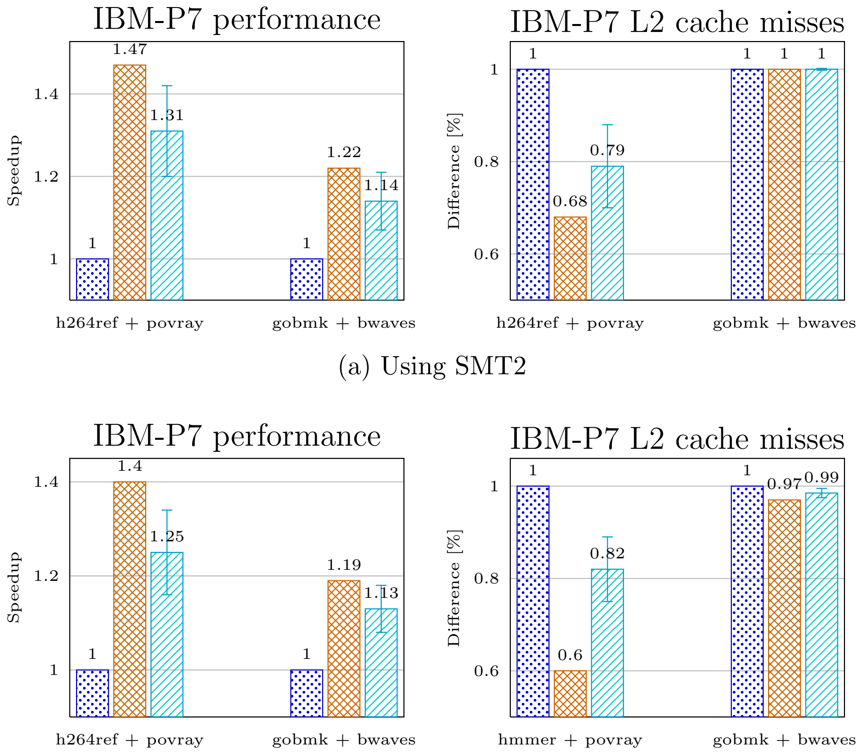
5.5. Multithreaded Spec.CPU on IBM-P7

In this experiment we repeat on IBM-P7 the same test previously performed on AMD-Bull in Section 5.4. Since IBM-P7 supports different levels of SMT, we use two additional configurations: SMT2 (reported in Figure 12a, using 2 PUs per core) and SMT4 (Figure 12b, activating 4 PUs per core). This means that for SMT2 (and accordingly to the machine settings described in Section 5.1) we run 8 instances of Spec.CPU integer and 8 instances of Spec.CPU floating point benchmarks, while for SMT4 we run 16 and 16.

We report and discuss here the results obtained with two pairs particularly significant and representative of the behaviours already pointed out on AMD-Bull in the previous section. The two pairs are *h264ref+povray* and *gobmk+bwaves*.

Since each IBM POWER7 processor core also shares its L2 cache among the available active PUs, we observe the number of L2 cache misses generated during the experiment to have a better explanation about the performance difference reasons.

Results. In general, this experiment confirms the results we obtained and discussed on AMD-Bull. WorkOver improves the worst-case baseline by 19% up to 47% and gives, on average, a boost of 6% to 16% over the default scheduler (in addition to more stable results). In fact, the performance variability observed on IBM-P7 by using the default OS scheduler is higher than the one measured on AMD-Bull. On IBM-P7, the top performance the OS scheduler can reach (reported as the highest point in the chart error bars) is lower than the one obtained on AMD-Bull (when both are compared to the



(a) Using SMT2

(b) Using SMT4

Figure 12: Spec.CPU speedups (top) and cache miss differences (bottom) on IBM-P7 with two different SMT configurations (SMT2 and SMT4), using an intentionally inefficient baseline (▒) versus using WorkOver to improve performance (▩) or letting the default OS scheduler do the job (▨).

performance obtained using WorkOver). The reason is that we are running more threads (16 and 32 on IBM-P7, respectively 8 on AMD-Bull) and it is more difficult for the default OS scheduler to obtain, by chance, a proper scheduling of the various threads.

According to our measurements, the speedups obtained on IBM-P7 are also in part due to a different cache usage and improved occupancy of the execution units. The *h264ref+povray* pair is a good case showing that the high difference in performance is followed by a proportional variation in the number of L2 cache misses. By comparing the speedup obtained by the WorkOver scheduling over the suboptimal baseline with the cache misses measured we observe that a speedup of 47% is related to 32% less misses (using SMT2) and a speedup of 40% to 40% less cache misses (using SMT4). If we check the cache miss rate in Table 2 for the benchmarks used in this pair, we discover that *povray* is the benchmark generating the lowest number of cache misses on IBM-P7. This fact suggests that threads running on the same core with threads running *povray* take more advantage from the shared L2 cache, since *povray* is not putting much stress on it (while all the other benchmarks have higher cache rates).

On the contrary, the *gobmk+bwaves* pair is a clear example of speedup that cannot be explained by a better cache exploitation. On SMT2, this pair shows a speedup of 22% over the suboptimal baseline and of 8% (on average) over the default OS scheduler, but the number of L2 cache misses measured in the three cases are identical. The SMT4 case is similar: a speedup of 19% over the baseline is followed by only 3% less cache misses, while a speedup of 6% over the default OS scheduler shows a variation of 1% in terms of misses.

5.6. Multithreaded SciMark2.0 on AMD-Bull

Thanks to Spec.CPU, we have been able to setup a case study by using two very distinct benchmarks running for a long period of time without significant variations. By switching to SciMark2.0, we want to use a more dynamic scenario where all of its 5 benchmarks are doing FPU-intensive computations with variable stress on the FPUs. Since SciMark2.0 runs inside a Java VM, WorkOver also keeps track of corollary threads that are used by the JVM for doing tasks such as realtime code optimization and garbage collection (thus providing a more noisy and realistic scenario).

Also, we modified the benchmark harness to start multiple threads concurrently executing thread-local instances of the benchmarks, in order to have persistent threads that change the benchmark they are running over

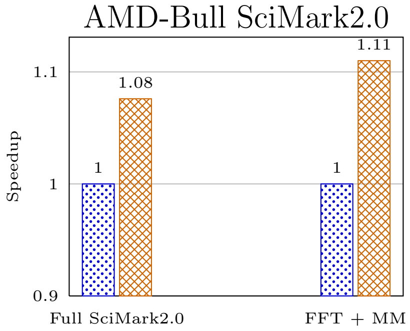


Figure 13: SciMark2.0 speedups on AMD-Bull using default OS scheduling (░) versus using WorkOver to improve performance (▩).

time (to really take advantage from the dynamic monitoring and adaptation of WorkOver). To saturate the PUs and cores we configured on AMD-Bull, we use 8 benchmark threads, that is, a number of threads equal to the number of available PUs. While the five SciMark2.0 benchmarks are executed in random order, the harness guarantees that each thread executes them all. After each thread completes the execution, we measure the cumulative score of each benchmark. Unlike Spec.CPU, SciMark2.0 benchmarks generate very few cache misses: for this reason, we do not show a similar chart for this test.

Results. Figure 13 reports the results of our evaluation, normalized to the baseline (i.e., default OS scheduling without WorkOver). Each datapoint corresponds to the average of 10 measurements. Each measurement is preceded by a warm-up run to attenuate noise from class-loading and just-in-time compilation. In all considered cases, the standard deviation is below 2%. When each benchmark thread executes the entire SciMark2.0 suite (*Full SciMark2.0*), WorkOver effectively improves runtime performance, incrementing the final benchmark score of 8%. Figure 13 reports the results of the execution of a subset of benchmarks, that is, FFT and MM (*FFT + MM*). According to Table 1, the FPU usage made by those workloads is the most different (99% and 86% respectively). In this case, the use of WorkOver increases the score of 11% since the system is less saturated than by running the full suite (with more threads with higher FPU usage) and takes an additional boost of 3% from the improved scheduling of threads with different stress on the FPUs.

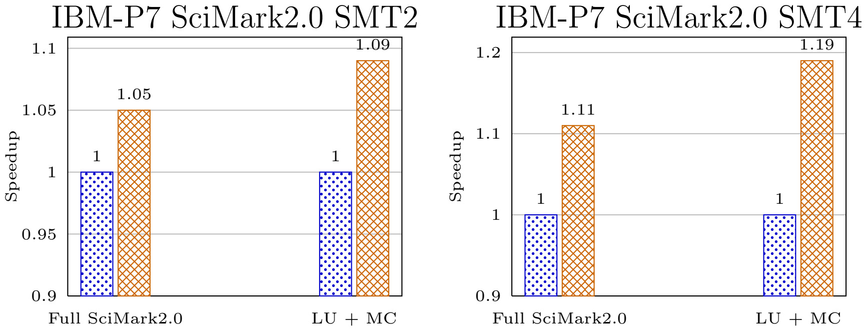


Figure 14: SciMark2.0 speedups on IBM-P7 with SMT2 and SMT4 using default OS scheduling (░ ░) versus using WorkOver to improve performance (▩ ▩).

5.7. Multithreaded SciMark2.0 on IBM-P7

In this experiment we execute on IBM-P7 the test we previously ran on AMD-Bull by using the same methodology described in Section 5.6 for the warm-up phase and to compute averages. In addition, as we already did in Section 5.5, we use two different SMT configurations to evaluate our approach under SMT2 and SMT4.

Results. Results are reported in Figure 14, normalized to the baseline defined as the average performance of the default OS scheduling after 10 runs. Standard deviation is below 5% for the baseline and below 2% for the results obtained using WorkOver. These results basically confirm the validity of our approach on a more noisy and dynamically changing scenario such as the one setup with multithreaded SciMark2.0.

On SMT2, we measure speedups of 5% for the full SciMark2.0 and of 9% for the pair *LU + MC*, that is, when only threads running the most and less FPU-intensive SciMark2.0 benchmarks are used (according to Table 2).

On SMT4, WorkOver is able to increase performance by 11% when the whole benchmark suite is used and up to 19% for the pair *LU + MC*. Since IBM-P7 runs on a different architecture and uses a different JVM, the code executed is very different from the one running on AMD-Bull. For this reason, the SciMark2.0 benchmark ratios reported in Table 2 show a higher variability (close to 50%) when compared to the ratios of Table 1 (with all the 5 benchmarks within a 15% of variability). This difference makes it more difficult for WorkOver to optimize the scheduling on AMD-Bull, while on IBM-P7 an improved usage of the available execution units provides a more significant speedup. In this case, too, WorkOver also produces more stable results over the baseline, by reducing the standard deviation from 5% to 2%.

5.8. Discussion

While on the one hand the approach we adopted in WorkOver has demonstrated an interesting series of results so far, it also features several limitations that must be discussed.

In the present work we intentionally avoided NUMA and inter-CPU constraints since it was beyond our scope. Nevertheless, NUMA-awareness is an important factor already largely analyzed in the literature than can bring higher speedup than the one provided by WorkOver when properly addressed. From this perspective, a two-level scheduling policy should be applied addressing first an efficient NUMA-aware distribution of the workload and then, within each NUMA-node (as we simulated in our experiments), improved execution unit occupancy.

WorkOver is also built on the assumption that heterogeneous integer and floating point workload is delivered to the machine. Many scientific applications do the same type of computations in all their threads, thus making our PU-oriented optimizations less effective. However, unless these threads are making a perfectly constant integer/floating point usage over time, WorkOver is capable of rescheduling them according to their current activity (as our experiments highlighted in Section 5.6 and 5.7).

Another factor that is currently not taken into account is data shared among threads: if two or more threads access the same information, scheduling them to distant cores without shared caches could reduce performance. This is an open issue that will be addressed by future work: when is it more worthy to improve for better execution unit occupancy and when for improved cache locality?

6. Related Work

Hardware performance counters represent a widely used instrument for performing realtime profiling of different computational workloads. Counters have been used for memory optimization [16], hardware characteristics identification [17], application characterization [18], security [19], data-race detection [20], etc.

The idea of exploiting counters for the development of hardware-aware scheduling policies has been already discussed in related research: in [21], for instance, hardware performance counters are exploited to drive the scheduling of multiple independent threads (i.e., not belonging to the same application) to reduce the power consumption on multicore machines. This and similar approaches [22, 23, 24] demonstrate how the fine-grained quality of the low-level counter measurements is of great benefit for performance optimization. In [25] the authors use cache-related performance data to enforce threads sharing a common data structure to share a common last-level cache. Similarly, in [26] the authors present a NUMA-aware scheduler based on performance counters. The scheduler monitors memory-related counters and infers which threads are sharing data on a common NUMA node. Therefore, the scheduler can easily map threads sharing the same resource to the most efficient NUMA node. The approach is specific to the domain of OpenMP parallel applications, while a generic approach is presented in [27], where a NUMA-aware scheduler has been introduced. The authors also show that schedulers not aware of the hardware architecture (called UMA systems in the paper) could even hurt performance. A similar non-NUMA approach has been presented in [28]. All these approaches show how advanced, hardware-aware scheduling policies improve performance in the case of hardware resource contention. In our work, we also observed the impact of physical resources contention, showing that execution units should also be carefully considered for thread scheduling, in addition to the evergreen cache and NUMA problems. An approach with similar goals is discussed in [29], where the benefits coming from different scheduling policies on an Intel-based HyperThreading-enabled multicore CPU are presented.

Klug et al. used hardware performance counters and thread-pinning [30] to improve performance through automatic detection of the best binding between threads and cores in a shared memory system. While their approach focused mainly on shared caches, with WorkOver we aim at improving performance through better execution unit occupancy.

Another relevant approach more specific to the IBM POWER7 architecture is presented in [31], where authors analyze how performance is influenced by thread placement policies. In particular, their analysis highlights that up to 54% reduction in execution time can be obtained (11.2% on average) when running parallel applications under the appropriate thread placement. In [32] thread placement is also analyzed with regard to energy consumption. Other related approaches are also discussed in [33, 34].

Finally, a discussion on the accuracy and the benefits of using different counters in measurement libraries and monitoring applications is presented in [35] and in [36].

7. Conclusion and Future Work

Modern micro-architectures are increasingly complex and heterogeneous with a growing adoption of SMT- and out-of-order-based solutions to provide a sustained stream of instructions to keep all the available processor execution units busy. In this article we present a case study for performance optimizations targeting shared hardware resources such as the ones found on the AMD Bulldozer and IBM POWER7 processors. In our experiments we show that a scheduler not aware of the underlying hardware characteristics incurs a significant performance penalty with threads featuring an emerging profile (integer- or floating point-intensive).

To address this limitation, we propose an approach based on monitoring, workload profiling and optimization assembled into WorkOver, a non-intrusive Linux-based tool running in user-space that allows to automatically and dynamically identify which threads are FPU-intensive and to schedule them in a more efficient way. Our measurements using two standard benchmarks (Spec.CPU and SciMark2.0) show that speedups of about 20% and more stable performance can be achieved by simply running WorkOver with the desired applications, without the need of any static analysis, source-code modification, nor changes to the default OS scheduler.

As future work, we want to evaluate our approach on other processor architectures (e.g., Intel Sandybridge and multicore ARM processors). We are also considering integrating the optimizations described in this article into a more sophisticated, multi-level scheduler, e.g., by using a hierarchy of rules that are first addressing NUMA and last-level shared cache aspects and then, within each NUMA node, optimized execution unit occupancy. From this perspective, one of the challenges will be the identification to the proper policy weight to give to each level. We also plan to investigate the power efficiency of our improved scheduling to see whether the speedup measured in our experiments also provides a more efficient energy consumption.

References

[1] D. Patterson, The trouble with multi-core, IEEE Spectr. 47 (7) (2010) 28–32.

[2] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[3] A. Peternier, D. Ansaloni, D. Bonetta, C. Pautasso, W. Binder, Hardware-aware thread scheduling: the case of asymmetric multicore processors, in: Proc. of the 18th International Conference on Parallel and Distributed Systems (ICPADS), Singapore, 2012.

[4] Z. Majo, T. R. Gross, Matching memory access patterns and data placement for numa systems, in: Proc. of the Tenth International Symposium on Code Generation and Optimization (CGO), ACM, 2012, pp. 230–241.

[5] D. Durand, T. Montaut, L. Kervella, W. Jalby, Impact of memory contention on dynamic scheduling on numa multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (11) (1996) 1201–1214.

[6] Z. Majo, T. R. Gross, A template library to integrate thread scheduling and locality management for numa multiprocessors, in: Proc. of the 4th USENIX conference on Hot Topics in Parallelism (HotPar), 2012, pp. 12–12.

[7] N. Min-Allah, H. Hussain, S. U. Khan, A. Y. Zomaya, Power efficient rate monotonic scheduling for multi-core systems, J. Parallel Distrib. Comput. 72 (1) (2012) 48–57.

[8] P. J. Nistler, J.-L. Gaudiot, Power efficient scheduling for hard real-time systems on a multiprocessor platform, in: Proc. of the 2010 IFIP international conference on Network and Parallel Computing (NPC), Springer-Verlag, 2010, pp. 106–120.

[9] J.-J. Chen, Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics, in: Proc. of the 2005 International Conference on Parallel Processing (ICPP), IEEE Computer Society, Washington, DC, USA, 2005, pp. 13–20.

[10] X. Zhao, N. Jamali, Fine-grained per-core frequency scheduling for power efficient-multicore execution, in: Proc. of the 2011 International Green Computing Conference and Workshops (IGCC), IEEE Computer Society, 2011, pp. 1–8.

[11] G. Anselmi, B. Blanchard, Y. Cho, C. Hales, M. Quezada, IBM Power 750 and 755 Technical Overview and Introduction REDP-4638-00, International Business Machines Corporation, 2010.

[12] J. Abeles, L. Brochard, L. Capps, D. DeSota, J. Edwards, B. Elkin, J. Lewars, E. Michel, R. Panda, R. Ravindran, J. Robichaux, S. Kandadai, S. Vemuganti, Performance Guide for HPC Applications on IBM Power 755, IBM Systems and Technology Group, 2010.

[13] J. Du, N. Sehrawat, W. Zwaenepoel, Performance profiling of virtual machines, in: Proc. of the 7th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE), 2011, pp. 3–14.

[14] A. Peternier, D. Bonetta, W. Binder, C. Pautasso, Overseer: Low-level hardware monitoring and management for java, in: Proc. of the 9th international conference on the Principles and Practice of Programming in Java (PPPJ), Denmark, 2011, pp. 143–146.

[15] M. W. Krentel, Libmonitor: A tool for first-party monitoring, Parallel Comput. 39 (3) (2013) 114–119.

[16] M. M. Tikir, J. K. Hollingsworth, Using hardware counters to automatically improve memory performance, in: Proc. of the ACM/IEEE conference on Supercomputing (SC), 2004, p. 46.

[17] J. Demme, S. Sethumadhavan, Rapid identification of architectural bottlenecks via precise event counting, SIGARCH Comput. Archit. News 39 (3) (2011) 353–364.

[18] Y. Luo, K. W. Cameron, Instruction-level characterization of scientific computing applications using hardware performance counters (wwc), in: Proc. of the Workload Characterization: Methodology and Case Studies, 1998, pp. 125–131.

[19] C. Malone, M. Zahran, R. Karri, Are hardware performance counters a cost effective way for integrity checking of programs, in: Proc. of the 6th ACM workshop on Scalable Trusted Computing (STC), 2011, pp. 71–76.

[20] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, T. Austin, Demand-driven software race detection using hardware performance counters, in: Proc.

of the 38th annual International Symposium on Computer Architecture (ISCA), 2011, pp. 165–176.

[21] K. Singh, M. Bhadauria, S. A. McKee, Real time power estimation and thread scheduling via performance counters, SIGARCH Comput. Archit. News 37 (2009) 46–55.

[22] S. Hsin-Ching, S. Bor-Yeh, Y. Wuu, L. Jenq-Kuen, Migrating java threads with fuzzy control on asymmetric multicore systems for better energy delay product, in: Proc. of the International Conference on Computing and Security (ICCS), 2011, pp. 1–12.

[23] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, E. Ayguade, Decomposable and responsive power models for multicore processors using performance counters, in: Proc. of the 24th ACM International Conference on Supercomputing (ICS), 2010, pp. 147–158.

[24] M. Y. Lim, A. Porterfield, R. Fowler, Softpower: fine-grain power estimations using performance counters, in: Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010, pp. 308–311.

[25] R. West, P. Zaroo, C. A. Waldspurger, X. Zhang, Online cache modeling for commodity multicore processors, SIGOPS Oper. Syst. Rev. 44 (2010) 19–29.

[26] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, B. R. de Supinski, Critical path-based thread placement for numa systems, in: Proc. of the 2nd international workshop on Performance Modeling, Benchmarking and Simulation of high performance computing systems (PMBS), 2011, pp. 19–20.

[27] S. Blagodurov, S. Zhuravlev, M. Dashti, A. Fedorova, A case for numa-aware contention management on multicore systems, in: Proc. of the USENIX Annual Technical Conference (USENIXATC), 2011, pp. 1–15.

[28] S. Blagodurov, S. Zhuravlev, A. Fedorova, Contention-aware scheduling on multicore systems, ACM Trans. Comput. Syst. 28 (2010) 8:1–8:45.

[29] J. Nakajima, V. Pallipadi, Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling, in: Proc.

of the 2nd Workshop on Industrial Experiences with Systems Software (WIESS), 2002, pp. 3–3.

[30] T. Klug, M. Ott, J. Weidendorfer, C. Trinitis, autopin: automated optimization of thread-to-core pinning on multicore systems, Transactions on high-performance embedded architectures and compilers III (2011) 219–235.

[31] S. Manousopoulos, M. Moreto, R. Gioiosa, N. Koziris, F. Cazorla, Characterizing thread placement in the ibm power7 processor, in: Proc. of the 2012 IEEE International Symposium on Workload Characterization (IISWC), San Diego, CA, USA, 2012.

[32] L. Brochard, R. Panda, S. Vemuganti, Optimizing performance and energy of hpc applications on power7, Computer Science - Research and Development 25 (2010) 135–140.

[33] D. Brelsford, G. Chochia, N. Falk, K. Marthi, R. Sure, N. Bobroff, L. Fong, S. Seelam, Partitioned parallel job scheduling for extreme scale computing, in: Job Scheduling Strategies for Parallel Processing, Vol. 7698 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 157–177.

[34] A. Vega, P. Bose, A. Buyuktosunoglu., Power-aware thread placement in smt/cmp architectures., in: Proc. of the 4th Workshop on Energy Efficient Design (WEED), Portland, OR, USA, 2012.

[35] D. Zaparanuks, M. Jovic, M. Hauswirth, Accuracy of performance counter measurements, in: Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009, pp. 23–32.

[36] S. Eranian, What can performance counters do for memory subsystem analysis?, in: Proc. of the ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC), 2008, pp. 26–30.