Hardware-aware thread scheduling: the case of asymmetric multicore processors

Achille Peternier, Danilo Ansaloni, Daniele Bonetta, Cesare Pautasso and Walter Binder
University of Lugano (USI), Switzerland
first.last@usi.ch

***Abstract*—Modern processor architectures are increasingly complex and heterogeneous, often requiring solutions tailored to the specific characteristics of each processor model. In this paper we address this problem by targeting the AMD Bulldozer processor as case study for specific hardware-oriented performance optimizations. The Bulldozer architecture features an asymmetric simultaneous multithreading implementation with shared floating point units (FPUs) and per-core arithmetic logic units (ALUs). BulldOver, presented in this paper, improves thread scheduling by exploiting this hardware characteristic to increase performance of floating point-intensive workloads on Linux-based operating systems. BulldOver 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 10% can be achieved by simply allowing BulldOver to monitor applications, without any modification of the workload.**

***Keywords*-multicore; workload characterization; asymmetric processors; performance**

I. 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 heterogeneity over symmetric 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 [2] (NUMA) controllers and hyperlinks, Simultaneous Multi-Threading (SMT) support, or ad hoc dedicated units. As a consequence, it is increasingly difficult for software developers to fully exploit the underlying computational power, as optimal software configurations can vary according to the hardware platform, to the application software architecture, and to the type of workload.

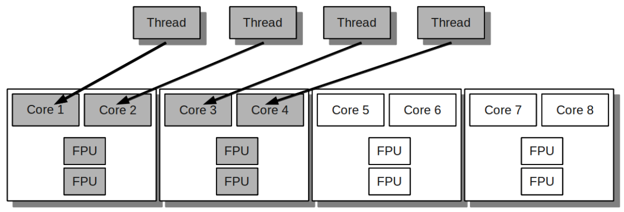
The Operating System (OS) kernel and scheduler try to optimize the performance of applications depending on the available hardware resources. To this end, OS schedulers rely on a limited set of performance indicators (such as CPU time and memory usage) to drive their optimization strategies. This is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

As a case study, in this paper we focus on one 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 sharing of hardware resources and their utilization.

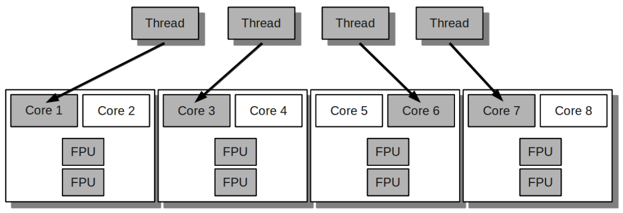
We focus on the AMD Bulldozer micro-architecture as it represents a good example of a modern hardware platform with specific characteristics that cannot easily be exploited by non-hardware-aware approaches. In this context, one of the peculiar characteristics of the Bulldozer architecture is the design of Floating Point processing Units (FPUs) which are shared between cores: two threads may contend for the same FPU unit. This hardware layout can have a negative impact on the performance of FPU-intensive workloads.

Our approach is named BulldOver (named after Bulldozer Overseer), and corresponds to a Linux daemon that interacts with the OS scheduler to improve thread scheduling of floating point-intensive workloads on Bulldozer processors. BulldOver runs in user-space and is based on performance metrics commonly available without any modification of the OS kernel and the monitored applications. Our approach relies on hardware performance counters to detect which threads make floating point-intensive computations, and on improved thread scheduling by running the most FPU-intensive threads on cores with a low level of contention for FPUs. In this way, BulldOver provides a bottom-up optimization mechanism, based on automatic workload characterization at runtime through hardware performance counters and on hardware-aware allocation of resources. No further intervention is required, neither in the workload nor at the OS scheduler level. The tool is a system-wide user-mode daemon collecting information and applying optimization policies on the threads used by applications that have been started via a specific command.

This paper is structured as follows. In Section II we give an overview of our approach, followed by background on the AMD Bulldozer micro-architecture and on hardware per-



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



(b) Optimal scheduling: one thread per module. 8 FPUs used: each thread uses 2 dedicated FPUs.

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

formance counters (Section III). The design implementation of BulldOver is described in Section IV. Our approach is evaluated in Section V. Section VI discusses related work, while Section VII concludes.

II. 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 in a subset of its threads. A common OS scheduler would assign FPU-intensive threads to the available cores for execution, as it would do for any other application. The scheduler takes metrics such as the amount of consumed CPU time 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 a eight-core Bulldozer processor – where cores are in fact more sophisticated SMT units assembled into modules of 2 cores – could potentially result in an inefficient computing resources usage. If the OS scheduler assigns the 4 floating point threads to 4 cores belonging to two modules (see Figure 1a), the total number of FPUs used will be 50% less than when the same 4 threads are scheduled one per module (Figure 1b).

Our goal is to prevent such inefficient scheduling by identifying and binding threads performing floating point computations to specific cores. We rely on hardware performance counters to measure the number of FPU-related operations performed by each thread to dynamically identify which threads would benefit from our approach.

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), OS schedulers usually do not use them to perform a constant, system-wide monitoring of all the running threads because user-space applications would not be able to use additional counters for their own needs.

Our approach, based on a centralized monitoring daemon that applies optimization policies only to threads spawning 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. According to the number of floating point operations executed by each thread, BulldOver sorts the most FPU-intensive threads (up to the number of available Bulldozer modules) and binds them on a per-module basis, instead of per-core. Whereas other scheduling policies are based on other hardware resources (e.g., NUMA nodes) and target other purposes (e.g., power efficiency), in this work we focus on a different shared hardware resource (floating point processing units) to improve performance.

III. Background

A. AMD Bulldozer Processors

This section describes in details the internal architecture of the AMD Bulldozer processor family[1]. The CPU embeds multiple *processing modules*. Each module 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 module 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 unit.

As a consequence, a Bulldozer has two Arithmetic and Logic Units (ALUs) per core, 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 module and not per core (see Figure 2). Two threads performing floating point operations scheduled on the same module will share 2

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

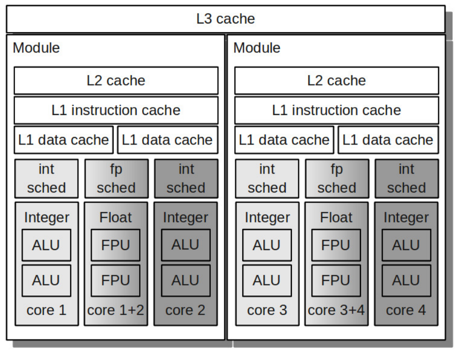


Figure 2: Architecture of a Bulldozer CPU with 2 modules and 4 cores.

FPUs, while the same threads scheduled one per module will use a total of 4 FPUs (2 FPUs for each thread exclusively).

The result is that Bulldozer modules 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[2]).

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

B. *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 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 [3], providing insight on what is happening at the hardware level.

In our case, we use counters to dynamically determine which threads (and in which percentage) execute floating point instructions. Based on this information, we schedule them on specific cores to make a better use of the Bulldozer hardware architecture.

IV. BULLDOVER DESIGN

To verify and translate our ideas into practice, we extended [4] into BulldOver, a tool for Linux-based systems written in C++ using hwloc[3] for inspecting the underlying hardware configuration, libpfm[4] to access performance counters, and libmonitor[5] for intercepting the creation and termination of threads and processes. BulldOver is designed with a client-server architecture (see Figure 3). 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 3, for instance, there are three applications being monitored. Applications to monitor (called Daemon Clients) are started just like any other Linux process through the shell prompt by preceding the application name with the *bulldover* command. This uses libmonitor to register callbacks that are invoked each time a new process or thread is created and terminated by the application. The advantage is that every application can be monitored without any change. Therefore, BulldOver is totally transparent to the processes being monitored.

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 BulldOver 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 process instance. This allows to obtain a fine-grain monitoring of the hardware performance events generated by each process (and by each thread spawning within each process, in the case of multithreaded applications).

The performance events are used to sort the list of the monitored threads to identify which threads are stressing more the FPU units. The control loop is closed by setting the CPU-thread affinity mask of the top FPU-intensive threads to bind them to the available modules. Non FPU- or less FPU-intensive threads are scheduled by the OS scheduler and are left untouched by BulldOver. The events monitored by BulldOver are the following:

- **PERF_COUNT_HW_CPU_CYCLES** measures 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. The 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 units are not being used.

Our assumption is that these two counters give an empirical estimate of the floating point load that each thread is generating. By polling these values regularly (every second in our experiments based on scientific workloads), BulldOver can dynamically identify which threads have recently been executing floating point-intensive workloads. Such threads

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

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

[4]http://perfmon2.sourceforge.net/

[5]https://outreach.scidac.gov/projects/libmonitor/

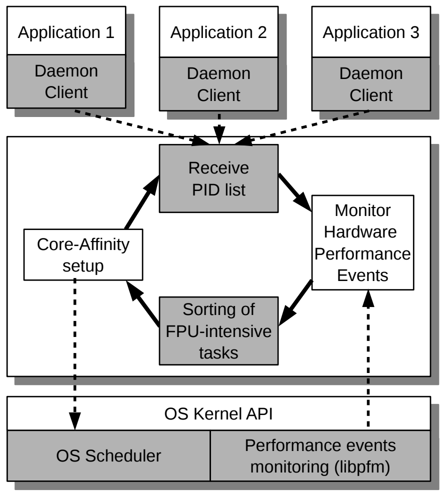


Figure 3: Adaptive control loop implemented by the BulldOver daemon.

will be privileged against threads not using the FPU units.

Like FPUs, L2 caches are also a resource shared by the two cores of one same Bulldozer's module. To differentiate the speedup contribution given by improved FPU usage from possible effects due to the L2 cache, we use an additional performance counter (L2_CACHE_MISSES) to measure the number of cache misses happening at that level.

V. Evaluation

A. Testing Environment

Tests are performed on a 4 CPU Dell PowerEdge M915 with 128 GB of RAM. Each CPU is an AMD 6282SE 2.6 GHz processor with 16 cores/8 modules. CPU frequency scaling and Turbo Mode have been disabled.

This machine features 8 NUMA nodes with 2 nodes per CPU[6]. To avoid latencies introduced by the non-uniform architecture, and since our work is not aiming at NUMA-aware scheduling, 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 8 cores/4 modules (as the one depicted in Figure 1) and 16 GB of RAM. As a consequence, all the cores share the same L3 cache. The OS is Ubuntu Linux Server 64bit version 11.10. C++ code is compiled using GCC version 4.6.1 for 64bit architectures with *-O3* optimizations. We used Oracle JDK 1.7.0_2 Hotspot Server VM 64bit.

Our evaluation approach is on realistic case studies and relies on two established benchmark suites: Spec.CPU[7] and SciMark2.0[8]. 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 choosen 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.

[6]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.

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

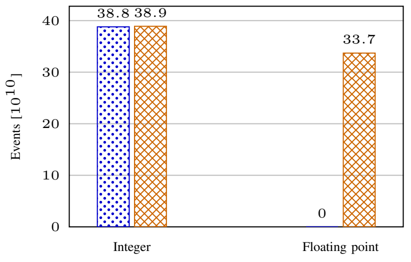


Figure 4: Counters microbenchmark showing empty FPU cycles (▨) and total CPU cycles (▨).

B. Workload characterization

To validate the usage of performance counters as an instrument for workload characterization and to verify the accuracy they provide in the identification of floating point intensive code, we first executed two synthetic experiments running two workloads: *Integer* and *Floating point*. For the first experiment, a series of mathematical operations is executed using only integer (in the *Integer* case) or floating point variables (*Floating point*). The execution of the workloads is monitored through the CYCLES_FPU_EMPTY and PERF_COUNT_HW_CPU_CYCLES counters previously described. Results are reported in Figure 4.

The CYCLES_FPU_EMPTY counter is effectively tracking the number of FPU-related operations. Each run of the test lasts approximately half a minute: 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 only contains FPU-related ones.

To confirm these results, we applied the same methodology to characterize the FPU workload generated by Spec.CPU and SciMark2.0 benchmarks. Results are reported in Figure 5 and Figure 6. We use these values to determine a ratio given by Equation 1.

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

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

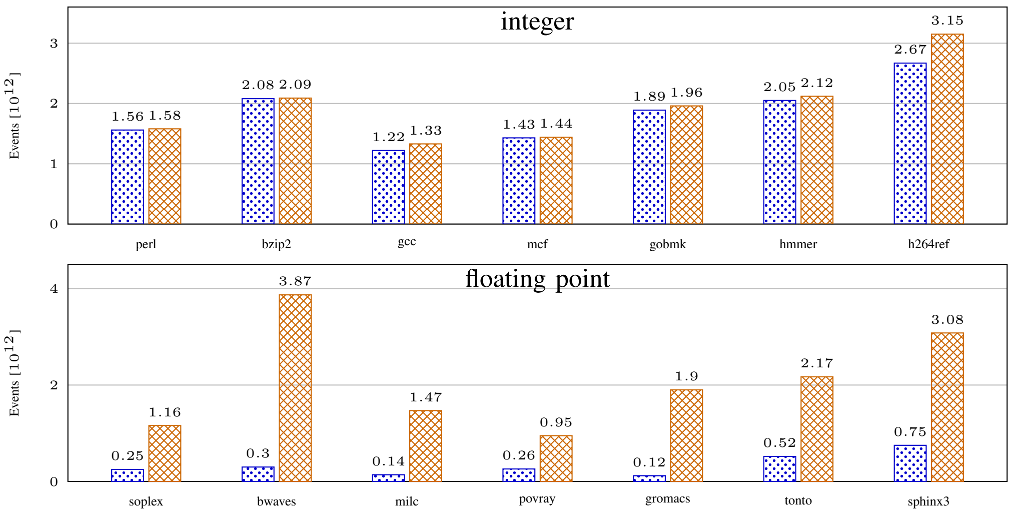


Figure 5: Empty FPU cycles (⬚) and CPU cycles (▩) count for each test of the Spec.CPU benchmark.

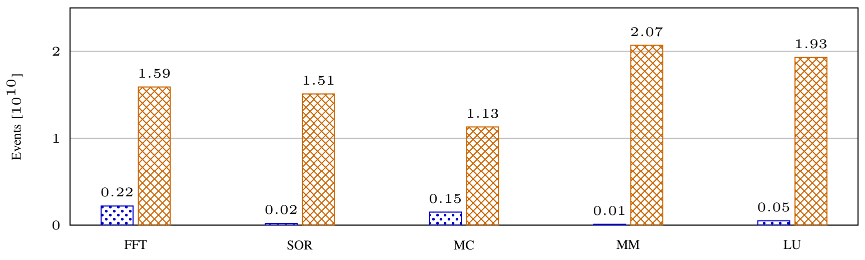


Figure 6: Empty FPU cycles (⬚) and CPU cycles (▩) count for each test of the SciMark2.0 benchmark.

We use *FPU usage* to characterize the FPU workload generated by each thread. Spec.CPU and SciMark2.0 FPU usages are shown in Table I. Thanks to this metric, and by regularly updating the values reported by performance counters, BulldOver monitors which threads are performing more FPU-intensive computations and forces the scheduler to assign them to a module where no other FPU-intensive threads are already running. This mechanism is at the basis of the controller used by BulldOver, which is evaluated in the next experiment. Table I is completed by an additional index: the L2 cache miss ratio. This ratio is defined by the number of CPU cycles divided by the number of L2 cache misses. We use this value to have a raw approximation of the level of contention that each benchmark put on the L2 cache. Since each module shares its FPUs and its L2 cache between its two cores, later on we will use this ratio to differentiate the performance contribution given by improved FPU usage and from better locality. SciMark2.0 benchmarks are very compact in terms of both memory consumption and code length, efficiently fitting within CPU caches. As a consequence, 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.

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 ratio						
0.12	0.5	0.84	1.62	0.08	0.07	0.07

FPU usage — *Spec.CPU floating point*						
soplex	bwaves	milc	povray	gromacs	tonto	sphinx
0.79	0.92	0.91	0.72	0.94	0.76	0.75
L2 cache miss ratio						
2.14	0.98	1.74	0.01	0.05	0.24	1.63

FPU usage — *SciMark2.0*				
FFT	SOR	MC	MM	LU
0.86	0.99	0.87	0.99	0.97

Table I: 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 ratios.

C. *Case Study*

In this section, we evaluate the speedup achievable with BulldOver when the system executes heavy numeric computations. When only a few threads are active, the OS is likely to schedule them on different modules. However, when the system utilization increases, threads executing floating point-intensive workloads may be scheduled on cores that are part of the same module. When using BulldOver, the system monitors the FPU usage ratio of all running threads. Every second, the most floating point intensive threads are bound to different modules, decreasing the contention on shared FPUs. In our testing environment using 8 cores/4 modules, this implies that the scheduling of the first 4 FPU-intensive threads is restricted to the first core of each module. The following 4 threads are scheduled on the second core of each module, while remaining threads have no restrictions. To reduce unnecessary thread migrations, a simple caching mechanism 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.

D. *Multithreaded Spec.CPU*

In this experiment we measure the wall-time required to run 4 instances of a Spec.CPU integer and 4 instances of a Spec.CPU floating point benchmarks. The idea is to automatically let BulldOver analyze and decide how to schedule them to improve the usage of FPUs. We include the results obtained with two pairs of benchmarks (*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 one same module (that is: integer with integer, floating point with floating point); (2) the optimized scheduling (▩) putting heterogeneous threads toghether (an integer thread with a floating point thread); and (3) by not using BulldOver at all and letting the default OS scheduler run (▨). Results are reported in the upper chart of Figure 7.

Since one Bulldozer module shares the FPUs and the L2 cache among its two cores, we also report (in the bottom 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 mostly 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 BulldOver 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 cores. To improve locality and prevent performance degradation due to unnecessary context switches, it tends to keep one thread tied to a specific core until the overall workload changes. In this way, in some cases the scheduler manages to distribute integer and FPU 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 BulldOver the optimal deployment is immediately identified and setup. The BulldOver controller also embeds a caching system that minimizes thread migrations: as long as one thread is considered a top consumer of either the integer or FPU group, it is permanently assigned to a specific core.

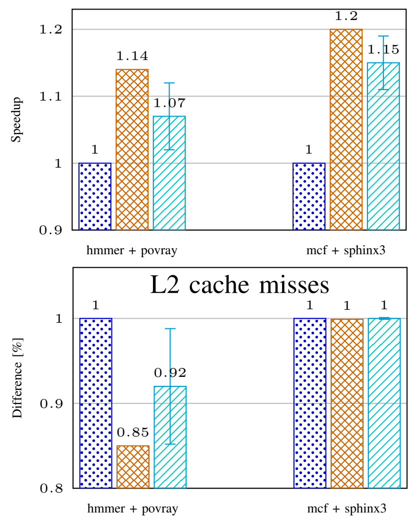


Figure 7: Spec.CPU speedups (top) and cache miss differences (bottom) using an intentionally inefficient baseline (▦) versus using BulldOver to improve performance (▩) or letting the default OS scheduler do the job (▨).

Results show that BulldOver is from 15% to 20% faster than the suboptimal baseline and by 6% to 10% faster (on average) than the default scheduler. BulldOver 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 BulldOver is concretely bringing some advantages to the system, but how can we be sure that this speedup is really coming from improved FPU usage and not from some other side-effects? According to our measurements, speedups are coming from both. The two pairs we selected are well representative of that: 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 ratio of Table I 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 the L2 cache is shared by the two cores, when we put two *hmmer* threads on the same module, 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 misses 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 ratio 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 FPUs.

E. Multithreaded SciMark2.0

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 more or less stress on the FPUs. Since SciMark2.0 runs inside a Java VM, BulldOver will also keep track of corollary threads that are used by the JVM itself 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 benchmark, in order to have persistent threads that change the benchmark they're running over time (to really take advantage from the dynamic monitoring of BulldOver). To saturate the single NUMA node we are using, we use 8 benchmark threads, that is, a number of threads equal to the number of available cores. While the five SciMark2.0 tests 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 test. 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 8 reports the results of our evaluation, normalized to the baseline (i.e., default OS scheduling without BulldOver). Each data point corresponds to the average of 5 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*), BulldOver effectively improves runtime performance, incrementing the benchmark score of 8%. Figure 8 reports the results of the execution of a subset of the workloads, that is, FFT and MM (*FFT + MM*). According to Table I, the FPU usage made by those workloads is the most different (99% and 86% respectively). In this case, the use of BulldOver 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.

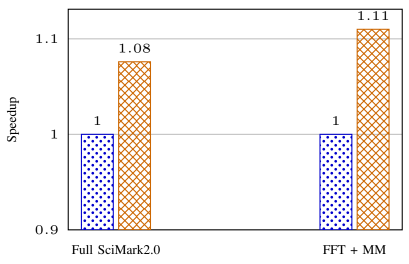


Figure 8: SciMark2.0 speedups using default OS scheduling (▨) versus using BulldOver to improve performance (▨).

VI. Related Work

Hardware performance counters represent a widely used instrument for performing realtime profiling of different computational workloads. Counters have been used in several different domains, including memory optimization [5], hardware characteristics identification [6], application characterization [7], security [8], data-race detection [9], etc.

The idea of exploiting counters for the development of hardware-aware scheduling policies has been already discussed in related research works: in [10], 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 [11], [12], [13] demonstrate how the quality of the low-level counter measurements is of great benefit for performance optimization. In [14] the authors use cache-related performance data to enforce threads sharing a common data structure to share a common last-level cache. Similarly, in [15] 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 [16], 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 [17]. All these approaches show how correct scheduling policies improve performance in the case of hardware resources contention. In our contribution we also demonstrated the impact of physical resources contention, showing how not only caches and memory contention should be carefully considered for thread scheduling, but also the contention of internal CPU modules such as FPUs. An approach with similar goals is discussed in [18], where the benefits of scheduling multiple threads on an Intel-based HyperThreading-enabled multicore CPU are presented.

Finally, a discussion of the accuracy and the benefits of different counters measurement libraries and approaches is discussed in [19] and in [20].

VII. Conclusion

Modern micro-architectures are increasingly complex and heterogeneous. In this paper we present a case study for performance optimizations targeting shared hardware resources such as the ones found on the AMD Bulldozer processor. The Bulldozer architecture features an asymmetric SMT implementation with FPUs shared by pairs of cores. In our experiments we show that a scheduling not aware of this characteristic incurs a significant performance penalty. To address this limitation, we propose an approach based on monitoring, workload characterization and optimization assembled into BulldOver, 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 standard benchmarks show that speedups of about 10% and more stable performance can be achieved by simply running BulldOver with the desired applications, without the need of any static analysis and source-code modification.

Acknowledgment: The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225) and by the European Commission (Seventh Framework Programme grant 287746).

References

[1] D. Patterson, "The trouble with multi-core," *IEEE Spectr.*, vol. 47, no. 7, pp. 28–32, 2010.

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

[3] J. Du, N. Sehrawat, and 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.

[4] A. Peternier, D. Bonetta, W. Binder, and 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 2011)*, Denmark, 2011, pp. 143–146.

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

[6] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 353–364, 2011.

[7] Y. Luo and 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.

[8] C. Malone, M. Zahran, and 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.

[9] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and 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.

[10] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, pp. 46–55, 2009.

[11] S. Hsin-Ching, S. Bor-Yeh, Y. Wuu, and 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.

[12] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and 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.

[13] M. Y. Lim, A. Porterfield, and 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.

[14] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online cache modeling for commodity multicore processors," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 19–29, 2010.

[15] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, and 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.

[16] S. Blagodurov, S. Zhuravlev, M. Dashti, and 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.

[17] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, pp. 8:1–8:45, 2010.

[18] J. Nakajima and 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.

[19] D. Zaparanuks, M. Jovic, and 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.

[20] 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.