Maximizing Power Efficiency with Asymmetric Multicore Systems

Asymmetric multicore systems promise to use a lot less energy than conventional symmetric processors. How can we develop software that makes the most out of this potential?

Alexandra Fedorova, Simon Fraser University

Juan Carlos Saez, Complutense University of Madrid

Daniel Shelepov, Microsoft

Manuel Prieto, Complutense University of Madrid

In computing systems, a CPU is usually one of the largest consumers of energy. For this reason, reducing CPU power consumption has been a hot topic in the past few years in both the academic community and the industry. In the quest to create more power-efficient CPUs, several researchers have proposed an *asymmetric multicore* architecture that promises to save a significant amount of power while delivering similar performance to conventional symmetric multicore processors.

An AMP (asymmetric multicore processor) consists of cores that use the same ISA (instruction set architecture) but deliver different performance and have different power characteristics. Using the same ISA on all cores means that you can run the same binary on all cores and not have to compile your code with a different compiler for each core type. This stands in contrast to heterogeneous-ISA systems, such as IBM's Cell or Intel's Larrabee, where the cores expose different ISAs, so the code must be compiled separately for each core type. Heterogeneous-ISA systems are not the focus of this article.

A typical AMP consists of several fast and powerful cores (high clock frequency, complex out-of-order pipeline, high power consumption) and a large number of slower low-power cores (low clock frequency, simple pipeline, low power consumption). Complex and powerful cores are good for running single-threaded sequential applications, because these applications cannot accelerate their performance by spreading the computation across multiple simple cores. Abundant simple cores, on the other hand, are good for running highly scalable parallel applications.

Because of performance/power trade-offs between complex and simple cores, it turns out to be a lot more efficient to run a parallel application on a large number of simple cores than on a smaller number of complex cores that consume the same power or fit into the same area. In a similar vein, complex and powerful cores are good for running CPU-intensive applications that effectively use those processors' advanced microarchitectural features, such as out-of-order super-scalar pipelines, advanced branch prediction facilities, and replicated functional units. At the same time, simple and slow cores deliver a better trade-off between energy consumption and performance for memory-bound applications that spend a majority of their execution time fetching data from off-chip memory and stalling the processor.

An SMP (symmetric multicore processor) includes cores of only one type: either the complex and powerful ones, as in the Intel Xeon and AMD Opteron processors, or the simple and lower-power ones, as in Sun's Niagara. So in terms of providing the optimal performance/power, an SMP is ideally suited for some applications but not all. Having cores of different types in a single processor enables optimizing performance per watt for a wider range of workloads. Having cores of different

types on an AMP enables us to employ *specialization* (i.e., we can use each type of core for the type of computation where it delivers the best performance/energy trade-off). Specialization enables maximizing the overall efficiency of the system and as a result delivers better performance per watt and per area.

Although single-ISA AMP systems are not yet being built, much of the literature investigating the properties of these systems has originated from major hardware players such as Intel^{1,7} and HP,^{5,6} indicating that within the industry there is interest in this architecture. Furthermore, existing SMP systems can be *configured to be asymmetric* if one wants to reduce the amount of power consumed by the processor. For example, configuring some of the cores of an SMP to run at a lower-than-maximum voltage and frequency (via the dynamic voltage and frequency scaling facilities available on most modern CPUs) makes the CPU consume less power and makes the system asymmetric. In that case it is crucial to understand how to get the maximum performance on this asymmetric system in order to minimize performance losses associated with running some cores at a lower frequency.

Given the emergence of AMPs, how can you best exploit them to maximize power efficiency (performance per watt)? Our goal is to shed light on some of the challenges software developers will likely face while trying to achieve this, and to provide some practical advice on how to maximize exploitation of AMPs as a result. To that end, we provide several examples demonstrating how to employ specialization on AMP systems in order to maximize performance. One area that will be of particular importance is the design of scheduling algorithms for operating systems that wish to take full advantage of AMPs. We also discuss our experience in investigating the design of such algorithms. We discuss a few surprising findings that we have discovered about the implementation of these scheduling strategies, which we think will be important to those who are developing or adapting an operating system for this upcoming class of processor and platform.

SPECIALIZATION ON AMPS

Efficiency of AMP systems could be improved using two kinds of specialization: the first caters to diversity in application-level parallelism; the second caters to microarchitectural diversity of the workload.

CATERING TO DIVERSITY IN APPLICATION-LEVEL PARALLELISM

Diversity in application-level parallelism refers to the two broad categories into which applications can be classified: scalable parallel applications and sequential applications. Scalable parallel applications use multiple threads of execution, and increasing the number of threads typically leads to reduced execution time or increased amount of work performed in a unit of time. Sequential applications, on the other hand, typically use only one or a small number of threads, and it is difficult to structure the computation to run efficiently in a multithreaded environment. In addition to purely parallel or purely sequential applications, there is a hybrid type, where an application might have phases of highly parallel execution intermixed with sequential phases.

These two types of applications require different types of processing cores to achieve the best trade-off in performance and energy consumption. Suppose we have a scalable parallel application with a choice of running it on a processor either with a few complex and powerful cores or with many simple low-power cores. For example, suppose we have a processor with four complex and

powerful cores and another area-equivalent and power-budget-equivalent processor consisting of 16 simple/low-power cores. Suppose further that each simple core delivers roughly half the performance of one complex core. (The numbers to estimate the conversion ratios of performance and power in complex vs. simple cores were obtained from Hill and Marty.⁴) We configure the number of threads in the application to equal the number of cores, which is a standard practice for compute-intensive applications. If we run this parallel application on the processor with complex cores, then each thread will run roughly twice as fast as the same thread running on the processor with simple cores (assuming that threads are CPU-bound and that synchronization and other overhead is negligible), but we can use only four threads on the complex-core processor vs. 16 threads on the simple-core processor. Since using additional threads results in a proportional performance improvement in this application, we get twice as much performance running on a simple-core processor as on a complex-core processor. Recalling that these two processors use the same power budget, we achieve twice as much performance per watt.

Contrast this to running a sequential application, which cannot increase its performance by using additional threads. Therefore, using a single thread, it will run twice as slowly on a simple-core processor as on a complex-core processor, meaning we get twice as much performance per watt running on the complex-core system. An experienced reader will observe that power consumption on a simple-core system for this single-application workload could be reduced by turning off unused cores. Unfortunately, it is not always possible to turn off unused cores completely, especially if they are located in the same power domain as the active cores. Furthermore, an operating-system power manager may be configured to avoid putting the unused cores in a deep sleep state, because bringing the cores up from this state takes time. Thus, if a new application begins running or if the operating system needs a core for execution, then additional latency will be incurred while the dormant core is being brought up to the active power state.

This example demonstrates that applications with different levels of parallelism require different types of cores to achieve the optimal performance-per-watt ratio. AMP systems offer the potential to resolve this dilemma by providing the cores of both types. Another advantage of having both “fast” and “slow” cores in the system is that the fast ones can be used to accelerate sequential phases of parallel applications, mitigating the effect of sequential bottlenecks and reducing effective serialization. Hill and Marty demonstrated that for parallel applications with sequential phases AMPs can potentially offer performance significantly better than SMPs, as long as sequential phases constitute as little as 5 percent of the code.⁴

CATERING TO MICROARCHITECTURAL DIVERSITY OF THE WORKLOAD

The relative benefit that an application derives from running on a fast core rather than a slow one depends on the microarchitectural properties of the program.^{2,5,6,11} Some programs are very efficient at using the CPU pipeline: they have a high amount of instruction-level parallelism, meaning that a processor can issue many instructions in parallel without running out of work. These programs show good locality of memory accesses. As a result they rarely access the main memory and thus rarely stall the processor. We refer to these programs as *CPU-bound*.

At the other extreme are programs that use the CPU pipeline very inefficiently. They typically have a high processor cache-miss rate and thus stall the CPU pipeline, because they have to wait while their data is being fetched from main memory. We refer to these programs as *memory-bound*.

(Note that this is not the same as an I/O-bound application, which often relinquishes the CPU when it must perform device I/O. A memory-bound application might run on the CPU 100 percent of its allotted time, but it would use the CPU inefficiently.)

CPU-bound programs use the hardware of fast cores very efficiently; thus, they derive relatively large benefits from running on fast cores rather than slow cores. Memory-bound applications, on the other hand, derive relatively little benefit from running on fast cores. Figure 1 shows some example speedup ratios of applications in the SPEC CPU2000 suite on an emulated AMP system. An SMP was used to emulate this AMP system using dynamic frequency scaling. The fast core was emulated by running a core at 2.3 GHz; the slow core was emulated by using the frequency of 1.15 GHz. Note that some applications experience a 2X speedup, which is proportional to the difference in the CPU frequency between the two processors. These are the CPU-bound applications that have a high utilization of the processor's pipeline functional units. Other applications experience only a fraction of the achievable speedup. These are the memory-bound applications that often stall the CPU as they wait for data to arrive from the main memory, so increasing the frequency of the CPU does not directly translate into better performance for them. For example, a memory-bound application, equake, speeds up by only 25 percent when running on the fast core.

For systemwide efficiency, it is more profitable to run CPU-bound programs on fast cores and memory-bound programs on slow cores. This is what catering to microarchitectural diversity of the workload is all about. Recent work from the University of California, San Diego and HP demonstrated that AMP systems can offer up to 63 percent better performance than can an SMP that is comparable in area and power, provided that the operating system employs a scheduling policy that caters to the microarchitectural diversity of the workload.⁶

ASYMMETRY-AWARE SCHEDULING

Employing specialization is the key to realizing the potential of AMP systems. Specialization on AMP systems will not be delivered by the hardware; it is up to the software to employ *asymmetry-aware scheduling policies* that tailor asymmetric cores to the instruction streams that use them most efficiently. A thread scheduler must be aware of the asymmetric properties of the system and assign threads to cores in consideration of the characteristics of both. In this section we report on our experience in designing and implementing such asymmetry-aware scheduling algorithms in a real operating system. We first describe an algorithm that caters to diversity in application-level parallelism and then an algorithm that caters to diversity in the workload's microarchitectural properties.

A SCHEDULER CATERING TO DIVERSITY IN APPLICATION-LEVEL PARALLELISM

The idea behind our *parallelism-aware (PA)* scheduler is simple: it assigns threads running sequential applications or sequential phases of parallel applications to run on fast cores and threads running highly scalable parallel code to run on slow cores. The following example demonstrates that the PA scheduling policy can achieve a much better system efficiency than an asymmetry-agnostic policy. We emphasize that the goal in using the PA policy is to maximize systemwide efficiency, not to improve performance of particular applications. As a result this policy will be inherently unfair: some threads will have a higher priority than others in running on fast cores. Implications of a policy that equally shares fast cores among all threads are demonstrated in our earlier study.⁹

Consider an asymmetric processor with one fast core and nine slow cores, where the fast core delivers approximately twice as much single-threaded performance as the slow core. Let's say that we run a workload of one sequential application and one parallel application, and the parallel application has nine threads. Under a naïve scheduling policy that equally shares fast and slow cores among threads, each thread will run on the fast core 10 percent of the time and on the slow core 90 percent of the time. (Note that existing operating-system schedulers would not share complex and simple cores equally. Distribution of time on different core types would be arbitrary. We assume a policy that shares cores equally to simplify the example.) To simplify comparison of different scheduling policies, we use as our performance measure the overall workload speedup relative to running all threads on slow cores the entire time. Under this naïve policy, each thread will speed up by $1.1\times$ relative to running on a slow core (to work this out, consider that each thread runs at a speed of $2\times$ for 10 percent of the time and at a speed of $1\times$ for 90 percent of the time), and the workload-wide speedup will also be $1.1\times$. Note that when computing the speedup for a parallel application we assume that the speedup for the entire application is close to the *average* speedup of its threads rather than the *aggregate* speedup—this assumption is reasonable if threads of a parallel computation are working on a common task as opposed to performing unrelated tasks requiring no inter-thread communication.

Under a PA policy, the single-threaded application will run on the fast core for the entire time, and the threads of the parallel application will run on slow cores. As a result, the single-threaded application will speed up by a factor of $2\times$, and the parallel application will have no speedup ($1\times$). The average speedup for the two applications will be $1.5\times$, or 40 percent better than under the naïve policy.

As another example, consider a parallel application with 50 percent of its code executing

sequentially. An asymmetry-agnostic scheduler may fail to assign the bottleneck sequential phase to run on a fast core, but a PA scheduler would make sure to accelerate it. Suppose the fast core runs twice as fast as the slow core: the PA scheduler would deliver up to 25 percent performance improvement to that application (see figure 2).

Figure 3 shows the performance of a number of parallel applications on an emulated AMP system with our implementation of a PA scheduler in OpenSolaris relative to the default asymmetry-agnostic scheduler in that operating system. To emulate AMP we use a real multicore system (AMD Opteron with 16 cores), and we emulate fast cores by using a high clock frequency (2.3 GHz) and slow cores by using a low frequency (1.15 GHz). The implementation of the algorithm and the experimental platform are described in more detail in a previous work.¹⁰ In this experiment we use four fast and 12 slow cores. The applications used in figure 3 are drawn from several benchmark suites, such as SPEC OpenMP 2001, PARSEC, MineBench, and NAS. RNA is a bioinformatics application performing RNA sequence matching.

The figure shows a variety of speedup values. Applications with significant sequential phases (40-60 percent) experience a performance improvement of up to 26 percent relative to an asymmetry-agnostic scheduler. Applications with small sequential phases (wupwise, for example) experience no speedup from the PA policy, because they do not stand to benefit from acceleration of sequential phases on fast cores.

Next we discuss two main challenges involved in delivering the benefits of the PA scheduling policy to applications: effectively detecting sequential and parallel phases and avoiding performance overhead that may result from cross-core migrations.

DETECTING SEQUENTIAL AND PARALLEL PHASES. The simplest way to detect parallel and sequential phases in an application is to use the runnable thread count as a heuristic. If an

application uses a large number of threads, then its runnable thread count will be high, and this application would be in a parallel phase. Conversely, an application with a single runnable thread would be in a sequential phase. The great property of the runnable thread count is that in modern multithreading environments it is visible to the operating system, because these systems map application-level threads to kernel-level threads. Therefore, by monitoring the runnable thread count an operating system can distinguish between parallel and sequential phases in applications.

Unfortunately, in some situations using the runnable thread count for detection of sequential phases might not work. In particular, an application could be running nonscalable code while still using a large number of runnable threads. We describe two scenarios where this might happen and discuss potential remedies.

In one scenario, an application might be susceptible to an external scalability bottleneck—for example, as a result of memory bandwidth contention. In this case the system memory bus is saturated, and using additional threads does not speed up the application because those threads do not contribute to useful computation. A sensible way to solve this problem is to reduce the number of threads used in an application to the point where the application operates at its peak efficiency. Essentially, this boils down to configuring the number of threads properly in a parallel application. Suleman et al. describe a technique called feedback-driven threading, which allows you to dynamically determine the optimal thread count for parallel applications.¹²

In another scenario, an application might be limited by internal scalability bottlenecks: for example, there might be a load imbalance where some threads do more work than others, or there might be synchronization bottlenecks where one thread executes the code in a critical section while other threads wait. When threads wait they may either *block*, relinquishing the CPU, or *busy-wait*, spinning on the CPU in a tight loop. If threads block, then the runnable thread count is reduced

FIGURE 3
Speedup with PA Scheduler

and any such reduction is visible to the operating system; but if threads busy-wait, sequential phases might be hidden from the operating system.

Whether the application uses blocking or busy-waiting depends on the implementation of the synchronization primitives, which are used to construct critical sections or barriers. Busy-waiting makes sense on a multiprocessor when the wait times are expected to be short. Blocking a thread is an expensive operation that should be avoided during short waiting periods. If the wait time is long, however, blocking is usually preferred so as to avoid wasting CPU resources. One of the most popular strategies used in synchronization libraries is the *adaptive* mode, where a thread busy-waits for a while and then blocks. On an AMP system, using the right synchronization waiting mode is crucially important. If busy-waiting is used, a PA scheduler would not be able to detect and accelerate the bottleneck serial phases.

Figure 4 shows the performance of selected OpenMP applications that could be configured to use either the adaptive or the busy-wait mode in the synchronization library. We show the performance under the PA scheduler relative to the default OpenSolaris scheduler. We use the same emulated AMP setup as described for figure 3, but in this case we configure the system with one fast core and 12 slow cores. For each application, we show the performance under the busy-wait and the adaptive modes.

As the figure shows, the PA scheduler can deliver significant performance improvements (up to 40 percent) for applications with large sequential phases but only if sequential phases are exposed to the scheduler via the adaptive synchronization mode. When the busy-wait mode is used, the scheduler is unable to detect sequential phases and accelerate them on the fast core of the AMP.

An alternative to using the adaptive mode would be to implement a new synchronization mode where the synchronization primitive explicitly notifies the scheduler when a thread begins to spin.¹⁰

Using this waiting mode will further ensure that spinning threads do not waste the resources of fast cores. When making changes to the synchronization library is not an option, however, the adaptive synchronization mode will do the job.

AVOIDING THE OVERHEAD. A more difficult challenge in implementing PA or other asymmetry-aware algorithms is to avoid the overhead associated with migrating threads across the cores. Any asymmetry-aware algorithm relies on cross-core migrations to deliver the benefits of its policy. For example, the PA algorithm must migrate a thread from a slow core to a fast core if it detects that the thread is executing a sequential phase.

Migrations are an essential tool of asymmetry-aware algorithms, but unfortunately they can be quite expensive. The AMP shown in figure 5a consists of several memory domains, as is usually the case with modern multicore processors. A memory domain is defined to contain cores that share an LLC (last-level cache). LLC is the last “line of defense” on the frontier between the CPU and the main memory. Thus, if the required data is not in the LLC, then the processor has to fetch it from the main memory, which takes hundreds of CPU cycles and slows down the computation considerably. In contrast, fetching data from an LLC takes only a few tens of processor cycles. Therefore, we want to minimize the number of accesses to the main memory and try to satisfy data requests from an LLC or other CPU caches as frequently as possible.

In figure 5a, the fast core is located in a different memory domain from the slow cores, so every time the scheduler migrates a thread to the fast core, the thread loses the data accumulated in the LLC of the slow core’s memory domain and must fetch the data from the main memory. (Depending on the implementation of the processor, the thread might have to fetch the data from its old LLC, not from memory, but that is still more expensive than fetching it from the LLC in its current

memory domain.) As we will show later, this can cause significant performance overhead.

Consider now figure 5b, which depicts a different AMP system where each fast core is located in the same memory domain as several slow cores. This architecture makes it easier for the scheduler to avoid cross-memory-domain migrations. In this case, a scheduler will try to migrate a thread to a fast core that is within the same memory domain as the slow core where the thread was previously running, thus enabling the thread to reuse the data in the LLC.

Figure 6 shows performance overhead experienced by several applications on two emulated AMP systems: one is a migration-unfriendly system configured like the system in figure 5a; and another is the migration-friendly system configured like the system in figure 5b, except that each domain has only three slow cores because of the limitations of the experimental hardware. Our PA scheduler is designed to be topology-aware, meaning that it attempts to avoid cross-memory-domain migrations whenever possible. We measured the overhead by making the designated fast cores on these two systems run at the same frequency as the slow cores—so no performance gains were to be expected from asymmetry-aware scheduling, but the overhead was still present, since our scheduler still migrated threads across cores “thinking” that the system is asymmetric.

Comparing the performance of applications under the PA scheduler and the default scheduler, we can find the migration-related performance overhead. In this case, performance degradation under the PA scheduler is equivalent to migration overhead. As figure 6 shows, performance overhead can be quite significant on a migration-unfriendly system, but it becomes negligible on a migration-friendly system coupled with a topology-aware scheduler.

In summary, a parallelism-aware scheduling policy can deliver real performance improvements on asymmetric hardware for parallel applications limited by sequential phases. The key is to configure

the synchronization library to “reveal” the sequential phases to the scheduler. To avoid cross-memory-domain migration overhead, AMP systems should be designed such that fast cores share a memory domain with some of the slow cores and combined with a topology-aware scheduler that minimizes cross-domain migrations.

A SCHEDULER CATERING TO MICROARCHITECTURAL DIVERSITY

Remember that the idea of catering to microarchitectural diversity of the workload is to assign CPU-bound threads (or phases of execution) to fast cores and memory-bound threads (or phases) to slow cores. Recall from figure 1 that CPU-bound code will experience a higher relative speedup running on fast vs. slow cores than memory-bound code, so scheduling it on fast cores is more efficient. Just like the PA policy, this policy will be inherently unfair: it may improve performance of some applications at the expense of others, but it will improve the efficiency of the system as a whole.

The biggest challenge in implementing such an algorithm is to classify threads or phases of execution as CPU-bound or memory-bound at scheduling time. Two approaches have been proposed in the research community to address this challenge. The first approach entails running each thread on cores of different types, registering the speedup obtained on a fast core relative to a slow core and using the resulting *relative speedup* as the measure for classifying the applications. In a scheduling algorithm, a thread with a larger relative speedup would be given preference to run on a fast core, and a thread with a lower relative speedup would be more likely to run on a slow core. Since this approach relies on direct measurement of relative speedup, we refer to it as the *direct measurement approach*.

A second approach, referred to as the *modeling approach*, is to model the speedup on a fast vs. slow core using a summary of an application’s runtime properties obtained either offline or online. Modeling is less accurate than direct measurement but does not require running each thread on each type of core, thus avoiding potential load imbalance and expensive cross-core migration (we elaborate on these issues later). In an effort to build an asymmetry-aware algorithm that caters to microarchitectural diversity, we have experimented with both methods.

The direct measurement approach manifested several performance problems. Consider a scenario where each thread must be run on each core type to determine its relative speedup. Given that a running thread may switch phases of execution (i.e., it may be doing different types of processing at different points in time), this measurement must be repeated periodically; otherwise, the scheduler might be operating on stale data. Since the number of threads will typically be larger than the number of fast cores, there will always be a high demand for running on fast cores for the purpose of remeasuring relative speedup. As a result, threads that are “legitimately” assigned to run on fast cores by the scheduling policy will observe undue interference from threads trying to measure their speedup there. Furthermore, having too many threads “wanting” to run on scarce fast cores may cause load imbalance, with fast cores being busy and slow cores being idle. When we used this direct measurement approach in an asymmetry-aware algorithm, we found that these problems made it difficult to deliver significant performance improvement relative to an asymmetry-agnostic scheduler.^{9,11}

The modeling approach involved predicting relative speedup on different core types using certain properties of the running programs. Since we were keen on evaluating this approach on real hardware (simulators put limitations on the length and number of experiments that can be

performed), we could experiment only with the asymmetry that was caused by the differences in the clock frequency of different cores. As a result, our relative speedup model was tuned to work for this specific type of asymmetric hardware. (This is the only type of AMP configuration available on today's commercial hardware.) At the same time, we do not see any fundamental reasons why our model could not be adapted to work on other single-ISA asymmetric systems.

Recall that the main factor determining how much speedup a program would obtain from running on a fast core is how memory-bound the program is. A good way to capture memory-boundedness is via a *memory reuse profile*, a compact histogram showing how well a program reuses its data.³ If a program frequently reuses the memory locations it has touched in the past, then the memory reuse profile will capture the high locality of reference. If a program hardly ever touches the memory values used in the past (as would a video-streaming application, for example), the memory reuse profile will capture that as well. Memory reuse profiles are so powerful that they can be used to predict with high accuracy the cache-miss rate of a program in a cache of any size and associativity. This is precisely the feature that we relied on in evaluating memory-boundedness of programs and building our estimation model.

Without going into much detail, in our scheduling system we associate a memory reuse profile with each thread. We refer to this profile as the *architectural signature*, since it captures how the program uses the architectural features of the hardware. The idea is that an architectural signature may contain a broad range of properties needed to model performance on asymmetric hardware, but for our target AMP system, using just a memory reuse profile was sufficient. Using that profile, the scheduler predicts each program's miss rate in the LLC, and using that miss rate, it estimates the approximate fraction of CPU cycles that this program will spend waiting on main memory. The scheduler can then trivially estimate the speedup that each program will experience running on a fast core relative to a slow core (see figure 7). Then the scheduler simply assigns threads with higher estimated speedups to run on fast cores and threads with lower estimated speedups to run on slow cores, making sure to preserve the load balance and fairly distribute CPU cycles. The resulting scheduler is called HASS (heterogeneity-aware signature-supported), and more details about its implementation are available in our earlier publication.¹¹

To evaluate how well the approach based on architectural signatures helps the scheduler determine the optimal assignment of threads to cores, we compare the resulting performance with that under the *best static assignment*. A static assignment is one where the mapping of threads to

cores is determined at the beginning of execution of a particular workload and never changed thereafter. The best static assignment is not known in advance, but can be obtained experimentally by trying all static assignments and picking the one with the best performance. The best static assignment is the theoretical optimum for our signature-supported algorithm, since it relies on static information to perform the assignment (the architectural signature) and does not change an assignment once it is determined.

Figure 8 shows the performance obtained using our signature-supported algorithm relative to the best static assignment. We show the overall performance for seven workloads. Each workload is constructed of four SPEC CPU2000 applications, two of which are memory-bound and two of which are CPU-bound. Each workload is executed on an emulated AMP system with two fast cores and two slow cores, so one single-threaded application is running on each core. The fast cores run at 2.3 GHz, and the slow cores run at 1.15 GHz. We used the AMD Opteron (Barcelona) system for this experiment.

The best static assignment always results in running the CPU-bound applications on the fast cores and the memory-bound applications on the slow cores. As figure 8 shows, the signature-supported algorithm is always able to match the best static assignment within a couple of percentage points. In that figure we also report the performance of the direct measurement approach mentioned earlier. It struggles to do as well as the signature-supported algorithm.

Although the signature-supported scheduler performs rather well, using it in a real scheduler involves an important challenge: obtaining the memory reuse profile needed for construction of the architectural signature. We are aware of two methods of obtaining the memory reuse profile: offline (used in our experiments) and online.

Using an offline method we can obtain a profile by running the application through a profiler

that is capable of monitoring and summarizing its memory-access patterns (for x86 executables we used the Pin binary instrumentation tool;⁸ this approach is described in more detail in our earlier work¹¹). The offline-generated profile must be somehow attached to the program binary (for example, by embedding it in the binary itself). Furthermore, it requires cooperation on the part of the developer, who must be responsible for generating the memory reuse profile for typical executions of the program.

The benefit of using offline-generated profiles is that they require no runtime overhead in the scheduling algorithm associated with their generation. The drawback of offline profiles is that it may be difficult to generate them accurately for multithreaded programs, and it may be tricky to account for different program phases. If a program changes its architectural properties according to the phase that it executes, one needs to associate multiple profiles with a single program. Furthermore, if the behavior of the program changes depending on the input, profiles obtained offline may turn out to be inaccurate.

The other method involves generating a memory reuse profile online. Although we have not experimented with the online approach ourselves, a group from the University of Toronto has recently proposed an efficient online method for generating accurate memory reuse profiles.¹³ This online approach would not have the same performance overhead as the direct measurement approach, because it would not require running each thread on each core type; a profile is collected while the thread is running on a single core and thus would not create the associated load imbalance.

Finally, there may be another promising approach for online estimation of relative speedup. Recall that our architectural signature method determines relative speedup based on the estimated LLC miss rates. Instead of estimating these miss rates from memory reuse profiles, we could also *measure* the miss rates online using hardware performance counters. Our preliminary results indicate that this is a fruitful method for determining relative speedup online. We believe that this method will be an effective way of estimating the relative speedup dynamically while avoiding the drawbacks of the direct measurement technique and the complexity associated with generation of memory reuse profiles.

SUMMARY

Asymmetric multicore systems promise to deliver higher performance per watt than conventional symmetric processors. To realize these benefits, however, the workload must have sufficient diversity (in terms of either parallelism or microarchitectural properties), and the operating-system scheduler must be designed to leverage this diversity on asymmetric hardware. Our experience with designing asymmetry-aware schedulers demonstrates that these algorithms can be feasibly implemented on real systems and used to unleash the potential of AMP systems without significant overhead.

ACKNOWLEDGMENTS

This research was supported by the National Science and Engineering Research Council of Canada through a Strategic Project Grant STPSC 356815-07, Sun Microsystems, and the Spanish government through Research Contract CICYT-TIN 2008/508 and Consolider Ingenio2010 2007/2011.

REFERENCES

1. Annavaram, M., Grochowski, E., Shen, J. 2005. Mitigating Amdahl's law through EPI throttling. *ISCA*.
2. Becchi, M., Crowley, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of Computing Frontiers*.
3. Berg, E., Hagersten, E. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
4. Hill, M. D., Marty, M. R. 2008. Amdahl's law in the multicore era. *IEEE Computer* 41(7): 33–38.
5. Kumar, R., Farkas, K. I., Jouppi, N., et al. 2003. Single-ISA heterogeneous multicore architectures: the potential for processor power reduction. In *Proceedings of the International Symposium on Microarchitecture (MICRO-36)*.
6. Kumar, R., et al. 2004. Single-ISA heterogeneous multicore architectures for multithreaded workload performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
7. Li, T., Baumberger, D., Koufaty, D. A., Hahn, S. 2007. Efficient operating system scheduling for performance-asymmetric multicore architectures. In *Proceedings of the Conference on Supercomputing*.
8. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., Hazelwood, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*.
9. Saez, J. C., Shelepov, D., Fedorova, A., Prieto, M. 2009. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. Submitted to *Transactions on Parallel and Distributed Systems*.
10. Saez, J. C., Fedorova, A., Prieto, M., Vegas, H. Submitted to a conference for blind review. Details omitted to preserve anonymity.
11. Shelepov, D., Saez, J. C., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S., Kumar, V. 2009. HASS: a scheduler for heterogeneous multicore systems. *ACM Operating System Review*, 43(2): 66-75.
12. Suleman, M. A., Qureshi, M. K., Patt, Y. N. 2008. Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on CMPs. *SIGARCH CAM*, 36(1): 277–286.
13. Tam, D. K., Azimi, R., Soares, L.B., Stumm, M. 2009. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ALEXANDRA FEDOROVA is an assistant professor of computer science at Simon Fraser University (SFU) in Vancouver, Canada. She earned her Ph.D. at Harvard University in 2006, where she completed a thesis on operating-system scheduling for chip multithreaded processors. Concurrently with her doctorate studies, Fedorova worked at Sun Microsystems Research Labs, where she investigated transactional

memory and operating systems. She is the lead inventor on two U.S. patents. At SFU, Fedorova co-founded the SYNAR (Systems, Networking and Architecture) research lab. Her research interests span operating systems and virtualization platforms for multicore processors, with a specific focus on scheduling. Recently she started a project on tools and techniques for parallelization of video games, which has led to the design of a new language for this domain.

JUAN CARLOS SAEZ obtained an M.S. degree in computer science from the Complutense University of Madrid and a B.S. degree in music from Teresa Berganza Professional Music Conservatory, both in 2006. He is now a Ph.D. student in computer science at the Complutense University. His research interests include energy-aware and cache-aware task scheduling on multicore and manycore processors. His recent activities focused on operating-system scheduling on heterogeneous multicore processors, exploring new techniques to deliver better performance per watt, and quality of service on these systems. His work is supported by the Spanish government through an FPU fellowship grant.

DANIEL SHELEPOV graduated from Simon Fraser University (SFU) in 2008 with a B.Sc. in computing science. During his senior year he led a research effort on heterogeneity-aware scheduling under supervision from Alexandra Fedorova. He is a member of the Internet Explorer team at Microsoft.

MANUEL PRIETO-MATIAS received his Ph.D. in computer science from the Complutense University of Madrid (UCM) in 2000. He is now associate professor in the department of computer architecture at UCM. His research interests lie in the areas of parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and on complexity-effective microarchitecture design. His current research addresses emerging issues related to chip multiprocessors, with a special emphasis on the interaction between the system software and the underlying architecture. He has co-written numerous articles in journals and for international conferences in the field of parallel computing and computer architecture. He is a member of the ACM, IEEE, and IEEE Computer Society.

© 2009 ACM 1542-7730/09/1100 \$10.00