Hardware Round-Robin Scheduler for Single-ISA Asymmetric Multi-core

Nikola Markovic[1,2](✉), Daniel Nemirovsky[1,2], Veljko Milutinovic[4], Osman Unsal[1], Mateo Valero[1,2], and Adrian Cristal[1,2,3]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{nikola.markovic,daniel.nemirovsky,osman.unsal, mateo.valero,adrian.cristal}@bsc.es

[2] Department of Computer Architecture, Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

[3] Spanish National Research Council (IIIA-CSIC), Barcelona, Spain

[4] School of Electrical Engineering (ETF), Univeristy of Belgrade, Belgrade, Serbia
vm@etf.bg.ac.rs

Abstract. As thread level parallelism in applications has continued to expand, so has relevant research on heterogeneous CMPs. Nowadays multi-threaded workloads running on CMPs are common case, but as the quantity of these workloads increase and as heterogeneous CMPs become more diverse, thread scheduling within an operating system will become ever more critical to maintaining efficient performance and system utilization. As a consequence, the operating system will require increasingly larger amounts of CPU time to schedule these threads effectively. Instead of perpetuating the trend of performing complex thread scheduling to the software, we propose a simple yet effective mechanism that can easily be implemented in hardware which outperforms the typical Linux OS scheduler as well as Fairness scheduler. Our approach fairly redistributes running hardware threads across available cores within OS scheduling quantum. It achieves an average speed up of 37.7 percent and 16.5 percent respectively compared to the Linux OS scheduler and state-of-the-art Fairness scheduling when running a multi-threaded application workloads.

Keywords: Hardware · Thread · Scheduling

1 Introduction

The relentless push in technology scaling driven by Moore's law has resulted in more transistors packed into a very small area. Computer architects responded by integrating many cores on the same die. Chip multiprocessors, or CMPs for short, are now the most common way to build high-performance microprocessors, for a variety of reasons. Large uniprocessors are no longer scaling in performance, because it is only possible to extract a limited amount of parallelism from a typical instruction stream using conventional superscalar instruction issue

© Springer-Verlag Berlin Heidelberg 2015
J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 122–134, 2015.
DOI: 10.1007/978-3-662-48096-0_10

techniques. In addition, one cannot simply ratchet up the clock speed on today's processors, or the power budget will become prohibitive. Compounding these problems is the simple fact that with the immense numbers of transistors available on today's microprocessor chips, it is too costly to design and debug ever-larger processors every year or two.

CMPs avoid these problems by filling up a processor die with multiple, relatively simpler processor cores instead of one huge core. The exact size of a CMP's cores can vary from very simple pipelines to moderately complex superscalar processors, but once a core has been selected the CMP's performance can easily scale across silicon process generations simply by stamping down more copies of the hard-to-design, high-speed processor core in each successive chip generation. In addition, parallel code execution, obtained by spreading multiple threads of execution across the various cores, can achieve significantly higher performance than would be possible using only a single core. While parallel threads are already common in many useful workloads, there are still important workloads that are hard to divide into parallel threads. The low inter-processor communication latency between the cores in a CMP helps make a much wider range of applications viable candidates for parallel execution than was possible with traditional, multi-chip multiprocessors; nevertheless, limited parallelism in key applications is the main factor limiting acceptance of CMPs in some types of systems. Nowadays, chip multiprocessors (CMPs) may be symmetric (SCMP), consisting of many cores of the same type, or asymmetric (ACMP), where cores may differ from one another with respect to their functionality and/or performance [5,13]. As is shown by a number of recent studies ACMPs are likely to outperform SCMPs for a fixed budget (area or power or both) [11,15]. Since it is well known that different workloads have different resource requirements, the benefits of ACMPs are intuitive.

The need for a scheduling algorithm arises from the requirement for CMP and ACMP systems to perform multitasking (executing more than one process or thread at a time). Scheduling is the method by which threads, processes or data flows are given access to system resources (e.g. processor time). This is usually done to load balance and share system resources effectively or to achieve a target quality of service. Parallel applications relying on multiple threads must be efficiently managed and dispatched for execution if the parallelism is to be properly exploited. Thus, dynamic thread scheduling techniques are of paramount importance in ACMP designs since they can make or break performance benefits derived from the asymmetric hardware or parallel software. Several thread scheduling methods have been proposed and applied to ACMPs. Most of these make use of online or offline profiling as well as sampling or estimation techniques to determine the optimum thread to core mapping (in relation to performance and/or power) whenever a specific event is detected or scheduling time quantum is completed [1,9,19] among others. Though these scheduling techniques include certain performance or energy efficiency gains, their broad application remains stifled due to scalability limitations, runtime overheads, and additional hardware requirements and complexities. Our goal is to develop a scheduling policy that can be used as a foundation upon which to build practical and scalable hardware scheduling designs in order to increase the performance capabilities of ACMPs.

This paper provides the following **contributions**:

– We propose a Hardware Round-Robin Scheduling (HRRS) policy which is influenced by Fairness Scheduling techniques thereby reducing thread serialization and improving parallel thread performance.
– We analyze and evaluate the performance of the HRRS policy on an ACMP and show that it lowers total execution time by 37.7 percent and 16.5 percent respectively compared to the state-of-the-art Linux OS scheduler and Fairness scheduler when running a multi-threaded application workloads.

2 Motivation

The exciting rise of asymmetric multi-core processors (ACMPs) has fostered a critical reevaluation of the traditional scheduling mechanisms in order to take full advantage of the new hardware resources in relation to the increasingly common thread level parallelism as well as in meeting certain system performance and power requisites. The operating system scheduler module orchestrates critical execution time junctures, selecting which jobs to be admitted next into the system and the next process to run. A technique known as fair-share scheduling is used by computer operating systems where CPU usage is equitably divided between system users or groups, in contrast to equal distribution among processes. The Linux OS scheduler, based on a fair-share scheduler strategy, is a process scheduler which was merged into the 2.6.23 release of the Linux kernel as its default scheduler [8]. It handles CPU resource allocation for executing processes aimed at maximizing overall CPU utilization as well as interactive performance. Operating systems may feature up to three distinct types of schedulers, a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short-term or CPU scheduler. The names suggest the relative frequency with which these functions are performed.

The third type of scheduler, the primary focus of this work, is the short-term commonly referred to as the CPU scheduler. It is responsible for determining which of the ready processes (loaded into the memory by the other schedulers) should be sent for execution and on which computational core. This decision takes place periodically at interrupt points caused principally by the clock, I/O events, or OS system level calls. In relation to the long and short-term schedulers, the short-term scheduler must make scheduling decisions much more frequently. Furthermore, the short-term scheduler can be preemptive or non-preemptive based on its ability to force processes off the CPU. The preemptive method depends on a programmable interval timer that invokes a kernel level interrupt handler which implements the scheduling algorithm. A key function involved in the CPU-scheduling decision is the dispatcher which gives control of the CPU to the process selected. This function involves the context switching, changing to user mode, and jumping to the proper location of a program once it is restarted.

The actual time it takes for the dispatcher to perform its job stopping one process and starting another is known as the dispatch latency typically requiring several thousands of cycles [12]. Since the dispatcher needs to analyze the program counter values, fetch instructions, and load data into the registers of the CPU every time a process switch occurs, minimizing the dispatcher latency should be a primary objective. Moreover, it is also important to avoid unnecessary context switches due to the fact that the processor remains idle for a period of time during context switches.

It has been shown by Van Craeynest et al. [18], that in a asymmetric multi-core system, a round robin scheduler using threads pinned to cores produces no speedup compared to a lighter symmetric multi-core system for most multithreaded benchmarks. This behavior is caused by barrier-synchronized multi-threaded workloads since the execution progress is limited by the slowest thread which has little meaning in a symmetric system, but is significant for asymmetric systems since the thread pinned to the simplest core will be the weakest link that all other threads will have to wait for at every barrier. Work-stealing workloads, in contrast, allows for idle large cores to steal work that would normally be run on the small cores so that the execution time isn't as constrained.

Therefore, in asymmetric multi-core systems, guaranteeing fairness is fundamental for improving performance of multithreaded workloads. Fairness, as defined by giving each software thread equal execution time on each core or allowing each thread to make equal progress, enables all threads to reach the barriers simultaneously, and has been sown to provide average performance improvements of 14 percent (and up to 25 percent) compared with a pinned scheduler [18] for the system configuration we are using.

3 Hardware Round-Robin Scheduling

In the previous section we have noted the importance and the impact scheduling fairness may have on the potential speedup that can be achieved from the parallelization of multi-threaded applications and multiprocess workloads on single-ISA asymmetric multi-cores. In the next two subsections we describe the proposed Hardware Round-Robin Scheduling (HRRS) policy and discuss its hardware implementation.

3.1 HRRS Algorithm

Figure 1 is used to illustrate the inner workings of the HRRS approach, we will assume a system composed of an x86 ACMP hardware containing one large out-of-order (OoO) core and three smaller and identical in-order cores. The operating system is provided an abstracted homogeneous hardware view comprised of four identical logical cores, which correlate to four identical hardware threads. The OS scheduler maps threads to the logical cores which enables the OS scheduling policies and implementation to be left unmodified. While the OS scheduler maps threads to the logical cores at every software-quantum or other interrupts, the

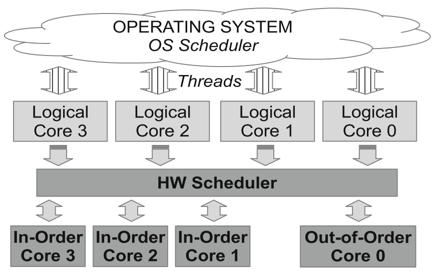

Fig. 1. HRRS scheduling - All logical cores (which correlate to hardware threads) are the same while the large physical core is represented by Core 0 and the small physical cores are shown as Cores 1,2 and 3

HRRS in turn maps the threads running on the logical cores to the physical cores as shown in Fig. 2 at every hardware-quantum. In essence, the HRRS can be viewed as mapping the logical cores that the OS sees and schedules threads onto, to the physical cores of the underlying hardware which actually execute the threads. Furthermore, the HRRS algorithm must produce a new scheduling scheme after every hardware-quantum of time passes (as opposed to the software-quantum which invokes the OS scheduler). In order to minimize the amount of overhead in implementing the scheduling policy, the HRRS algorithm determines the next scheduling scheme to apply before the beginning of the next hardware-quantum. The defining characteristic of the HRRS algorithm is that it evenly rotates threads (scheduled onto the logical cores by the OS scheduler) running on the physical cores after every hardware-quantum. The OS scheduler, on the other hand, is triggered at every software-quantum which happens much less frequently than that of the hardware-quantum. Additionally, the HRRS algorithm does not need to take into account whether the OS scheduler has activated and swapped one of the currently executing threads on a logical core for another thread from its ready queue. In such cases, the thread context of the thread being swapped out must be saved and replaced by the context of the new thread chosen by the OS to be executed all of which is performed by the triggered OS scheduler routine. Consequentially, the HRRS scheduling policy guarantees that a thread will not occupy a large physical core for more than one hardware-quantum unless it is the only runnable thread at the end of the hardware-quantum.

The Fundamental Difference. Between the HRRS and Fairness-aware scheduler algorithms [18] is the way in which the threads are selected to be mapped onto the physical cores. In both approaches, a thread running on one of the smaller physical cores is swapped with the thread running on the large physical core after a given time quantum. However, while Fairness-aware scheduling strives at achieving fairness by guaranteeing even progress using specific heuristic for each software thread, it does not necessarily enforce swaps of threads between large and small core every scheduling quantum but prefers to leave threads to run on the same physical core. In contrast, the HRRS policy runs each logical core,

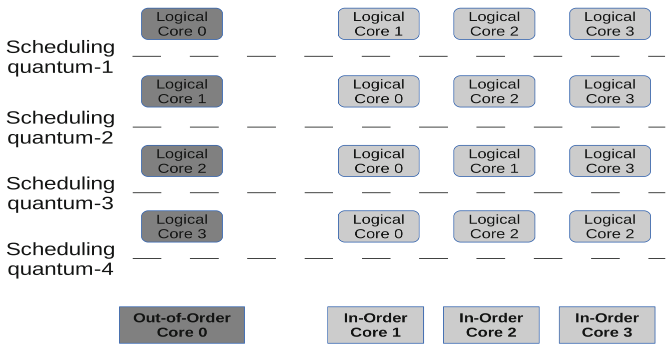

Fig. 2. An example of the HRRS scheduling logical cores on actual physical cores at every hardware-scheduling quantum. At the beginning logical core 0 is running on the large physical core while logical cores 2, 3 and 4 are running on small physical cores. After a first hardware scheduling quantum, Logical core 1 will be moved to large physical core and logical core 0 will be moved to a small physical core.

hardware thread, on each physical core type for a specified amount of time. After every quantum, the HRRS triggers a swap between the thread running on the large core with one executing on a small core that is chosen using a round-robin selection algorithm.

3.2 Hardware Implementation

Hardware Round-Robin Scheduling leaves the operating system level scheduling untouched and it maintains a consistent view of the underlying hardware. The hardware is able to provide the abstraction of a symmetric hardware to software while dynamically rescheduling threads among the cores in an asymmetric multi-core system [14]. Both of these approaches (HRRS and Fairness approach [18]) may also be implemented at the OS level by extending the OS scheduler but the advantage of a hardware approach, in addition to minimizing scheduling overheads, is that it provides a finer level of granularity for the scheduling quanta and requires no changes to the OS code [19].

HRRS has hardware additions which include a bit on every core to signal if the core is executing kernel or user code and a separate unit with vector that holds all of the bits, one counter and one decoder to facilitate round-robin mechanism. The size of these depends on the number of the cores in the system. For instance, for four core system we need 2-bit counter. Unlike to some of the other dynamic schedulers [7], the HRRS scheduling technique does not facilitate hardware overheads in order to be able to store and restore the architecture state in the cores. It utilizes the x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [6]. In the case that large core is in the kernel mode (handling an interrupt etc.) HRRS scheduler will wait until it returns to user mode to mark rescheduling, while if the small core that isto have its thread swapped with the large core is in the kernel mode, the scheduler will chose next small core to switch threads with large core in the round-robin fashion.

4 Evaluation

Here we evaluate the Hardware Round-Robin Scheduling (HRRS) approach and compare it to the Linux OS scheduler and Fairness approach [18].

4.1 Simulated Architecture and Workloads

For conducting the simulation experiments in this paper we have used Sniper [2], a parallel, hardware-validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications. We configured the simulator to model an ACMP made up of one large core and three small cores respectively. The differences between the core types lie in the pipeline complexity (out-of-order for large, in-order for small). In order to isolate the causes of potential performance differences, the clock frequency (2.6 GHz), issue width (4-wide), number of available thread contexts (one hardware context per core), and cache sizes are the same for both core types. We assume a cache hierarchy with separate and private 32 KB L1 instruction and data caches, private 256 KB L2 caches, and a shared 4 MB L3 last-level cache (LLC). All the caches employ a LRU replacement policy and we assume the memory controllers are on-chip. Similar to the work in [18], we utilize a conservative hardware-quantum of 1 ms and a software-quantum of 4 ms even though it is typically upwards of this range.

We use the SPLASH-2 [20] benchmarks in our experiments. The SPLASH-2 benchmarks are designed to represent multi-threaded applications in order to evaluate hardware architectures when running several thread contexts. All applications are run from start to finish. We run each benchmark on the four simulated cores with each core capable of executing one hardware thread context at a time. We evaluate single multi-threaded application workloads running an equal number of threads per application as the number of available hardware contexts, i.e., maximum number of threads, which is a common practice for running non-I/O-intensive applications [7]. For example, when we run one multi-threaded application on a system simulated with 4 cores, we use 4 threads for that application.

4.2 Performance Evaluation

Implementing dynamic scheduling requires the migration of workloads between different cores. This leads to overheads incurred by context switches and the loading of the working-sets into the private caches of the destination cores. A context switch incurs a fixed cost for storing and restoring the architecture state (at most a few kilobytes) [12] for which we presume a fixed 1,000 cycle penalty. Our simulations also take into account the warming of the cache hierarchy needed after a

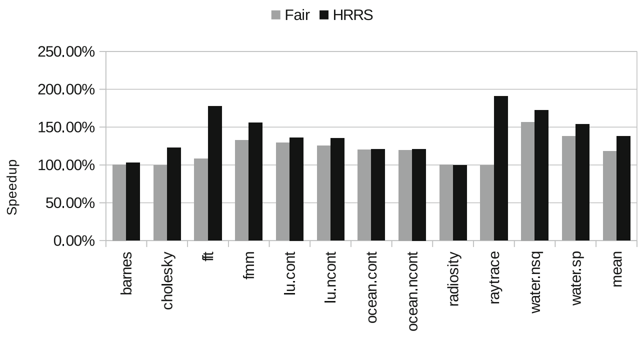

Fig. 3. Net Speedup comparison of the HRRS and Fairness scheduler normalized to Linux OS scheduler for the SPLASH-2 benchmark suite running on four cores (1 OoO + 3 InO)

context switch. The study [19] has shown the total migration execution time overhead to be less than 1.5 percent across different types of single-threaded workloads, ranging form memory-intensive to compute-intensive, for a 4 MB shared LLC using a 1ms hardware-quantum.

Figure 3 shows the speedup of the HRRS scheme over the hardware implementation of the Fairness scheduler on four core system running multi-threaded application workloads. The results are scaled to a Linux OS scheduler where the operating system has a notion of the underlying hardware. Under the Linux OS scheme, in the case of a symmetric CMP, the operating system pins individual threads to each of the cores in a round-robin fashion until all threads are assigned. The threads are then selected to be executed in a round-robin fashion on the respective core that they are pinned to (when there are more than one thread assigned per core). When using Linux OS scheduler threads on an asymmetric CMP, the operating system does not necessarily pin the threads to the cores, nor does it tend to swap the threads running on the large core with those on the small core at every scheduling quantum until threads pinned to the lagre core are all stalled or finished its execution. Rather, the OS scheduler tries to ensure quality-of-service for the threads. This reflects the current practice in contemporary operating system schedulers, as exemplified in the Linux 2.6 kernel [8]. The average speedups of HRRS over the Fairness and Linux OS scheduler when running the Splash2 workloads are 16.5 percent and 37.7 percent respectively on a four core system.

A key element driving these performance benefits comes from the redistribution of the workloads amongst the cores. Figure 4 shows per benchmark LLC access distribution between large core and small cores, while total number of the LLC accesses grows up by only up to 1.5 percent for Fair and HRRS schedulers compared to Linux OS scheduler. The HRRS scheme produces a higher proportion of LLC accesses originating from the large core. Fundamentally, the large core can better support the extra burden of LLC cache accesses since the large out-of-order instruction window allows for a greater quantity of instructions to be processes concurrently, which enables it to hide the additional latency caused by the extra

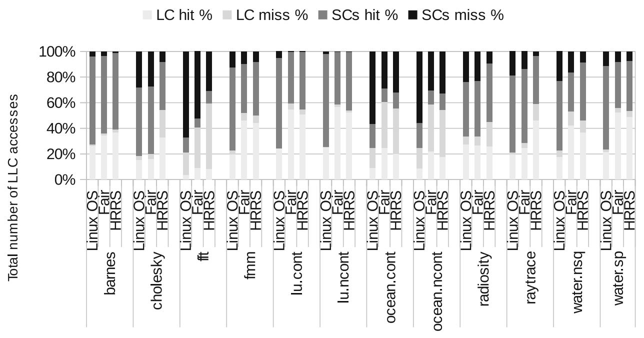

Fig. 4. The LLC cache accesses breakdown for the large and small cores of the Linux OS, Fairness and HRRS scheduler for the SPLASH-2 benchmark

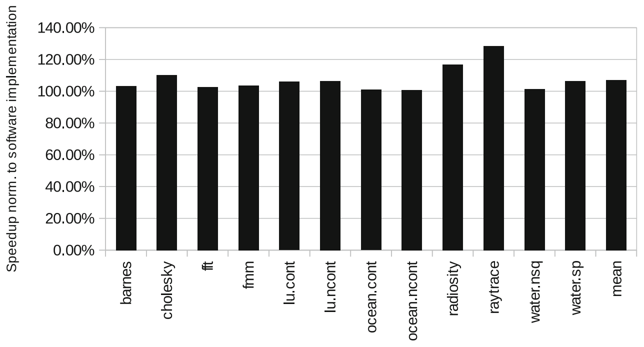

Fig. 5. Speedup of the hardware over the software implementation (baseline) for the HRRS scheduler, where scheduling quanta are 1 ms and 4 ms for hardware and software implementations respectively

cache accesses and still, even after including overheads from the context swap, outperform the small cores in thread execution time. This hit/miss ratio and considerable change in total number of LLC accesses between large and small cores are clearly noticeable with cholesky and raytrace benchmarks which have the highest performance gains.

Hardware vs. Software Implementation. Hardware Round-Robin Scheduling leaves the operating system level scheduling untouched and it maintains a consistent view of the underlying hardware. The hardware is able to provide the abstraction of a symmetric hardware to software while dynamically rescheduling threads among the cores in an asymmetric multi-core system [14]. Figure 5 represents the speedup that a hardware implementation, with a scheduling quantum of 1 ms, has over a software implementation (baseline), with a scheduling quantum of the 4 ms, for the HRRS scheduler. We can see that a hardware implementation results in an average speedup of 6.98 percent over a software implementation for the HRRS scheduler.

5 Related Work

Due to possible performance and efficiency gains, there has been increasing interest in heterogeneous multi-core architectures, and various scheduling proposals have been presented. An ACMP which consists of multiple cores of the same ISA but of different sizes was proposed by Kumar et al. in [10]. Their process consists of sampling for and choosing the core that will execute in the most power efficient manner each time a new phase or program is detected. This work was later expanded to maximize performance of multithreaded applications [11].

Similar work by Becchi [1] consists of an ACMP that includes two distinct core sizes where thread to core assignment is managed by initiating a mandatory swap of threads between two different sized cores in order to measure the corresponding performance ratio. Based on this ratio, the threads are then scheduled to their core that will maximize the system performance. This work has given insight into ratio based ACMP scheduling techniques but is limited as the number of distinct core types used increases. Other work in this area has been done by Saez et al. [16] who use a utility factor, defined as the ratio of L1 miss latency compared to a baseline ACMP configuration (only small cores), with the aim of optimizing the performance of both single and multithreaded workloads. Likewise, Koufaty et al. [9] determine optimal thread to ACMP core mapping using a biasing method estimated by the quantity of external memory stalls and internal pipeline stalls. Another approach is detailed in the work by Srinivasan et al. [17] who propose a formula based ACMP thread to core scheduling method which is used to estimate and compare thread performance on individual cores. With respect to microarchitectural differences, Chen et al. [3] chose to implement their ACMP with cores consisting of separate branch predictor, issue width, and L1 cache sizes that together with their scheduling method, achieve throughput and energy efficiency improvements.

6 Future Work

When considering an LLC cache for many-core processors, an popular option gaining traction in the industry is to distribute the cache into separate blocks, therefore appearing as unified rather than being physically unified. This is a similar approach to that taken by the IBM Power8 architecture [4] where each core has an 8 MB low-latency LLC cache and a high-speed cache-coherent ring is used to connect all of the cores. Thus joined, the LLC cache blocks can be viewed as a shared 96 MB cache with a nonuniform latency. Access to the local 8 MB LLC is speedy, but access to remote LLC cache blocks will require additional cycles to traverse the ring. By comparison, a large unified LLC cache would have a constant access time slower than the local cache but faster than remote blocks. The IBM design keeps hot data in the CPUs local LLC, reducing the average LLC latency.

In a many-core processor with this kind of distributed LLC cache configuration, a latency problem may arise due to frequent context switches among cores being connected to a different LLC cache segments. Therefore, the HRRS scheduling heuristic many need to be adjusted to account for the added latencies of the LLC and moreover can be tuned to allow for the scheduling of threads to be such as to take advantage of the distribution of cache blocks. Perhaps it may become viable to not only swap threads from large to small cores but also from small to small depending on which LLC segments their data sets are located. Furthermore, it would be beneficial to implement the HRRS scheme on an FPGA in order to gauge the feasibility of the design as well as raise the level of accuracy concerning the latency overheads.

7 Conclusion

In this paper we have presented the Hardware Round-Robin Scheduler (HRRS). Our work is influenced by the rise of many core processors, particularly the asymmetric core multi-processors (ACMPs) and their dependence on dynamic schedulers such as the commodity Linux OS CPU scheduler in order to achieve fair and balanced performance between active threads. Our initial objective was to achieve these performance benefits from running parallel workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our HRRS policy promotes a balanced distribution of execution time for threads per core type. We have shown that HRRS provides greater opportunity for all threads to share time running on the more efficient large core, selected via a round-robin algorithm, which produces generous performance benefits even after including scheduler and context swap overheads as well as latencies arising from the additional cache accesses needed for loading the working data sets. By using the HRRS policy on an ACMP, we got a total execution time speedup of 37.7 percent and 16.5 percent respectively compared to the state-of-the-art Linux OS scheduler and Fairness scheduler when running a multi-threaded application workloads (Splash2).

Acknowledgment. Nikola Markovic is supported through a doctoral scholarship by the Universitat Politecnica de Catalunya (UPC). Additional support is provided by the CICYT (Spanish National Science Foundation) project TIN2012-34557. We would also like to thank to all reviewers for their comments and suggestions.

References

1. Becchi, M., Crowley, P.: Dynamic thread assignment on heterogeneous multiprocessor architectures. J. Instr.-Level Parallelism **10**, 1–26 (2008)
2. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2011)

3. Chen, J., John, L. K.: Efficient program scheduling for heterogeneous multi-core processors. In: Proceedings of the Annual Design Automation Conference, pp. 927–930 (2009)
4. Fluhr, E.J. et al.: IBM STG, POWER8TM: a 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth. In: Proceedings of the IEEE International Solid-State Circuits Conference on Digest of Technical Papers, pp. 96–97 (2014)
5. Greenhalgh, P.: big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 (2011). www.arm.com/files/downloads/bigLITTLE_Final_Final.pdf
6. Intel Corp.: Intel 64 and ia-32 architectures developers manual (2015). http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html
7. Joao, J., Suleman, M.A., Mutlu, O., Patt, Y.: Utility-based acceleration of multithreaded applications on asymmetric CMPs. In: Proceedings of the Annual International Symposium on Computer Architecture, pp. 154–165 (2013)
8. Jones, M.T.: Inside the Linux 2.6 Completely Fair Scheduler (2009). http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf
9. Koufaty, D., Reddy, D., Hahn, S.: Bias scheduling in heterogeneous multi-core architectures. In: Proceedings of the 5th European Conference on Computer Systems, pp. 125–138 (2010)
10. Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., Tullsen, D.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture, p. 81 (2003)
11. Kumar, R., Tullsen, D., Ranganathan, P., Jouppi, N., Farkas, K.: Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of the Annual International Symposium on Computer Architecture, p. 64 (2004)
12. Li, C., Ding, C., Shen, K.: Quantifying the cost of context switch. In: Proceedings of the Workshop on Experimental Computer Science, p. 2-es (2007)
13. NVIDIA: Variable SMP: A Multi Core CPU Architecture for Low Power and High Performance (2011). http://www.nvidia.com
14. NVIDIA: Tegra 3 (Kal-El) Quad-Core Mobile Processor (2011). http://www.nvidia.com/object/tegra-3-processor.html
15. Rodrigues, R., Annamalai, A., Koren, I., Kundu, S., Khan, O.: Performance per watt benefits of dynamic core morphing in asymmetric multicores. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 121–130 (2011)
16. Saez, J.C., Prieto, M., Fedorova, A., Blagodurov, S.: A comprehensive scheduler for asymmetric multicore systems. In: Proceedings of the 5th European Conference on Computer Systems, pp. 139–152 (2010)
17. Srinivasan, S., Zhao, L., Illikkal, R., Iyer, R.: Efficient interaction between OS and architecture in heterogeneous platforms. SIGOPS Oper. Syst. Rev. **45**(1), 62–72 (2011)
18. Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A., Eeckhout, L.: Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of the International Conference on Parallel Architectures Compilation Techniques, pp. 177–187 (2013)

19. Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., Emer, J.: Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: Proceedings of the Annual International Symposium on Computer Architecture, pp. 213–224 (2012)
20. Woo, S., Ohara, M., Torrie, E., Singh, J., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the Annual International Symposium on Computer Architecture, pp. 24–36 (1995)