Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors

SHOAIB AKRAM, Ghent University
JENNIFER B. SARTOR, Ghent University and Vrije Universiteit Brussel
KENZO VAN CRAEYNEST, Ghent University
WIM HEIRMAN[1] , Intel Corporation
LIEVEN EECKHOUT, Ghent University

While hardware is evolving towards heterogeneous multicore architectures, modern software applications are increasingly written in managed languages. Heterogeneity was born of a need to improve energy-efficiency; however, we want the performance of our applications not to suffer from limited resources. How best to schedule managed language applications on a mix of big, out-of-order cores, and small, in-order cores is an open question, complicated by the host of service threads that perform key tasks such as memory management. These service threads compete with the application for core and memory resources, and garbage collection (GC) must sometimes suspend the application if there is not enough memory available for allocation.

In this paper, we explore concurrent garbage collection's behavior, particularly when it becomes critical, and how to schedule it on a heterogeneous system to optimize application performance. While some applications see no difference in performance when GC threads are run on big versus small cores, others — those with *GC criticality* — see up to an 18% performance improvement. We develop a new, adaptive scheduling algorithm that responds to GC criticality signals from the managed runtime, giving more big core cycles to the concurrent collector when it is under pressure and in danger of suspending the application. Our experimental results show that our GC-criticality-aware scheduler is robust across a range of heterogeneous architectures with different core counts and frequency scaling, and across heap sizes. Our algorithm is performance and energy-neutral for GC-uncritical Java applications, and significantly speeds up GC-critical applications: by 16% on average, while being 20% more energy-efficient for a heterogeneous multicore with three big cores and one small core.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles - Heterogeneous (hybrid) systems; D.3.4 [**Programming Languages**]: Processors - Memory management (garbage collection); D.4.1 [**Operating Systems**]: Process Management - Scheduling

General Terms: Design, Algorithms, Performance, Energy

Additional Key Words and Phrases: Managed runtimes, Concurrent collection, Automatic memory management, Heterogeneous multicore architectures, Energy-efficiency, GC criticality

ACM Reference Format:
Akram, S., Sartor, J. B., Van Craeynest, K., Heirman, W., Eeckhout, L. 2015. Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 24 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

[1]This work was completed while the author was at Ghent University.

This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295, as well as EU FP7 Adept project number 610490.
Author's addresses: S. Akram, J. B. Sartor, K. Van Craeynest, and L. Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: Shoaib.Akram@elis.UGent.be, Jennifer.Sartor@elis.UGent.be, Kenzo.VanCraeynest@elis.UGent.be, Lieven.Eeckhout@elis.UGent.be. W. Heirman, Intel Corporation, Veldkant 31, 2550 Kontich, Belgium; email: Wim.Heirman@intel.com.

1. INTRODUCTION

Managed languages running on top of runtime environments offer increased software productivity and portability. One key reason why managed languages are used in a broad spectrum of domains, ranging from data centers to handheld mobile devices, is that they offer automatic memory management through garbage collection (GC). Garbage collection reduces the chance of memory leaks and other memory-related bugs, while easing programming. However, garbage collection introduces overhead to the application's execution time [Cao et al. 2012], in part because managed language applications allocate objects rapidly [Blackburn et al. 2004; Zhao et al. 2009]. Garbage collection can be run in either a "stop-the-world" mode, where the application's progress is stopped while collection occurs, or in a "concurrent" mode, where the application and GC run at the same time. However, concurrent collection threads must coordinate and share resources with the application. Moreover, if the allocation rate exceeds the rate of collection, the application can run out of allocation space, which requires the application to be stopped while GC frees memory. This can lead to a large performance penalty.

On the hardware side, heterogeneous multicores have emerged because of the need for energy-efficient computing [Kumar et al. 2003, 2004]. Industry examples of single-ISA heterogeneous multicores include ARM's big.LITTLE [Greenhalgh 2011], NVidia's Tegra [NVidia 2011], and Intel's QuickIA [Chitlur et al. 2012]. These systems contain a mix of cores that vary in their ability to exploit instruction-level parallelism (ILP) and memory-level parallelism (MLP). Big cores that run instructions out-of-order exploit ILP and MLP by having many instructions in flight at the same time, usually achieving the best performance. Small cores that execute instructions in order provide a low-power alternative, and are limited in the amount of ILP and MLP that they can exploit. Heterogeneity provides a power-performance tradeoff, giving the ability to select the core that best matches the software's characteristics, within performance and energy constraints. However, dynamic scheduling of diverse workloads remains a challenging problem.

A significant body of recent work emphasizes the importance of scheduling on single-ISA heterogeneous multicores [Becchi and Crowley 2006; Chen and John 2009; Ghiasi et al. 2005; Koufaty et al. 2010; Lakshminarayana et al. 2009; Li et al. 2007, 2010; Shelepov et al. 2009; Srinivasan et al. 2011; Van Craeynest et al. 2012, 2013]. However, managed runtime environments include several service threads, such as garbage collection, that run for a significant fraction of the execution time [Cao et al. 2012; Du Bois et al. 2013b], and should be treated differently than application threads, according to recent research [Cao et al. 2012; Heil and Smith 2000; Hu and John 2006; Maas et al. 2012; Sartor and Eeckhout 2012]. Previous work [Cao et al. 2012] argues that because GC threads are not on the critical path, are memory-bound, and do not exhibit ILP, they should be scheduled on small cores in a heterogeneous multicore for the best performance per energy.

In this paper, we explore the behavior of concurrent garbage collection on big versus small cores for Java applications, aiming to optimize total application performance. Running benchmarks in the Jikes Research Virtual Machine (RVM) on top of a multicore simulator, we find that some applications, particularly multi-threaded applications with higher thread counts, are more garbage collection intensive, and benefit significantly if GC is run on big versus small cores, by as much as 18%. These benchmarks exhibit *GC criticality* during execution when the concurrent GC threads cannot keep up with application allocation, and thus GC threads must pause application progress and divert to a stop-the-world mode to collect memory. For other applications, however, we observe no performance difference when running GC threads on big versus small cores. In particular, single-threaded and some multi-threaded applications at small thread counts do not exercise GC much, and we call them *GC-uncritical*. To verify the generality of GC criticality, we also compared the performance of Jikes' best-performing production collector, stop-the-world generational Immix, when it runs on big versus small cores. Several benchmarks still benefit from running on out-of-order cores, as they demonstrate a performance difference of up to 15%. We conclude that GC criticality can occur in many different system setups. GC criticality is a function of a number of factors, including processor architecture, virtual machine, garbage collection algorithm and implementation, heap size, application characteristics, etc. The bottom line is that if garbage collection is unable to keep up with the application's memory allocation rate (because GC is receiving too few resources), garbage collection will become critical.

Based on these insights, we design a new, adaptive scheduling algorithm that responds to signals from the managed language runtime about GC criticality, which dynamically varies during the run, boosting GC threads' priority on the big core(s) only if GC is in danger of not keeping up with application allocation. Our GC-criticality-aware scheduler adapts to phase behavior, balancing performance and energy efficiency by lowering GC threads' priority on the big core(s) if GC becomes uncritical. While our scheduler is performance-neutral for GC-uncritical benchmarks, it improves performance significantly for GC-critical applications (compared to prior best practice which puts GC threads always on small cores [Cao et al. 2012]). Using a set of Java benchmarks from the DaCapo benchmark suite [Blackburn et al. 2006] on top of the Jikes Research Virtual Machine 3.1.2 [Alpern et al. 2000], we report an average performance improvement of 2.9%, 7.8%, and 16% for the GC-critical benchmarks when running on a four-core system with one, two, and three big cores, respectively, while at the same time improving energy-efficiency by 3.5%, 10.7% and 20%. Compared to an existing fair scheduler [Van Craeynest et al. 2013] which strives at achieving fairness across all runnable threads, our GC-criticality-aware scheduler achieves significantly better performance, especially for architectures with limited big core resources. We comprehensively evaluate the robustness of GC-criticality-aware scheduling across core counts, big to small core ratios, heap sizes, and clock frequency settings, and conclude that GC-criticality-aware scheduling is particularly beneficial as GC becomes more critical. GC-criticality-aware scheduling improves overall application performance by giving sufficient resources to GC so it can keep up with the application. We make the following contributions:

— We demonstrate that, contrary to prior work, garbage collection can significantly benefit (up to 18%) from out-of-order versus in-order execution by exploiting ILP.
— We pinpoint when GC becomes critical to overall application performance, namely when a concurrent collector cannot free memory fast enough for application allocation.
— Motivated by the observation that applications exhibit different sensitivities with respect to GC criticality, we propose an adaptive scheduling algorithm that receives semantic information from the memory manager about GC criticality, adjusting GC's priority for big core time slices, even taking slices away from the application so as to avoid costly stop-the-world pauses.
— We evaluate our adaptive scheduling algorithm, showing that it performs well across a large range of heterogeneous architectures and heap sizes. While our GC-aware scheduler is performance and energy-neutral for GC-uncritical applications, we see substantial performance and energy efficiency improvements for GC-critical applications.

This work shows that scheduling modern workloads on heterogeneous multicores significantly benefits from semantic information (GC criticality) provided by the managed runtime, in order to provide high performance on future energy-efficient processor architectures.

2. BACKGROUND

Before discussing garbage collection on heterogeneous multicores, we first provide background on managed languages and different kinds of garbage collection.

2.1. Garbage Collection

Managed languages have a range of service threads that perform runtime environment tasks. Such service threads include the dynamic compiler, profiling threads, and those that do memory management. While previous work [Blackburn and McKinley 2008; Yang et al. 2011] has tackled optimizing several of these tasks, recent research [Cao et al. 2012; Du Bois et al. 2013b] reveals that memory management continues to contribute a significant portion of the total execution time, because of excessive object allocation [Blackburn et al. 2004; Zhao et al. 2009].

The memory manager, or garbage collector (GC), provides regions of free memory to the application for fresh allocation, and automatically detects unused parts of memory and reclaims them to be used again. Garbage collection has a space-time tradeoff. The more heap space you give the application, the longer until a collection must be invoked. However, when the collection does occur because the heap is full, it could have more memory to trace, depending on the lifetime of objects, and thus take longer. On the other hand, if heap space is limited, garbage collection must be performed frequently, and yet, each collection does not last as long because the amount of live data is bounded.

Garbage collection involves tracing the reachability graph of the heap. The collector first identifies *roots* from which to trace, including addresses from the stack, globals, and statics. The collector maintains a list of addresses to be traced, and collector threads process that list by following pointers that point into the heap. When a heap object is found, it is marked as "live", and is searched for pointers to other heap objects, which are added to the list. Tracing is complete when there are no more elements on the processing list. All heap objects that are then not marked as "live" are unreachable by the application, and thus are freed to be used again.

Because garbage collection must look at all pointers to heap objects, it must see a consistent view of the heap. Thus, the easiest way to implement a garbage collector is by stopping the application completely during tracing and freeing. This is called *stop-the-world (STW)* collection. However, stopping the application completely while the whole heap is scanned causes long application pauses, which are undesirable, especially for interactive and real-time applications.

2.2. Concurrent Garbage Collection

Concurrent garbage collection runs garbage collection threads alongside application threads to reduce pause times. In this work, we consider the Jikes Research Virtual Machine's (RVM) [Alpern et al. 2000] concurrent collector, which is a traditional mark-sweep snapshot-at-the-beginning concurrent GC algorithm, based on Yuasa's algorithm [Yuasa 1990]. Figure 1 depicts the phases of an application that has four application threads (*a0* to *a3*) running with such a collector. This collector has four threads, with *g0* and *g1* running concurrently with the application, and *g2* and *g3* running when the application is stopped.

Most concurrent collectors require a small pause to the application to first identify a consistent root set (shown in Figure 1 as "roots"), and later to actually free memory (shown as "release"). In our concurrent collector, separate threads are spawned to perform the STW phases of collection (threads *g2* and *g3*). The traversal of the object graph can happen in parallel with the application (shown as the action of threads *g0* and *g1*) as long as newly allocated or modified objects are marked as "live" so that they are not freed by the collector. In addition, all application writes go through a barrier to coordinate with GC threads so that they are not writing to the same object, and so the GC maintains a consistent view of heap pointers [Blackburn and Hosking 2004]. Our concurrent collector initiates a new collection cycle (defined as starting with the "roots" phase, and ending with the "release" phase) after the previous cycle ends and if a parameter-defined quantity of memory in bytes has been allocated.

While pauses of the application are minimized when using a concurrent garbage collector, the application execution can still be stalled. If the application runs out of memory to allocate into, it must pause until garbage collection frees up enough memory for it. Jikes' collector then transitions into a stop-the-world mode (shown on the right in Figure 1 as the "scan" phase that makes collection slower). This STW pause can have a large performance cost, especially because bookkeeping work must be performed to transition from the concurrent to the STW mode, and switching threads could also cause cache perturbation.

2.3. Garbage Collection on Heterogeneous Multicores

While some prior work [Cao et al. 2012; Hu and John 2006] has explored the behavior of managed language services, including garbage collection, on heterogeneous cores, they have focused on optimizing energy. They found that GC can be put on a smaller core, or a scaled-down big core, in

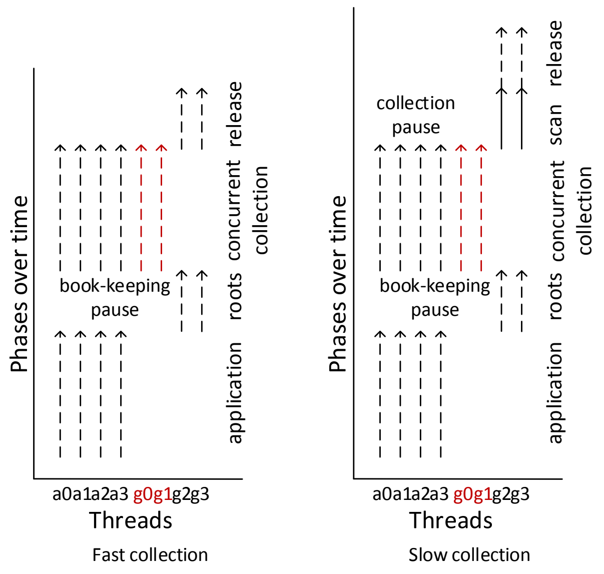


Fig. 1: Threads and phases of application execution using Jikes' concurrent collector, with the optional 'scan' pause (right) if concurrent GC threads cannot keep up with application allocation.

order to save energy. Both prior works argue that GC does not have instruction-level parallelism, and uses a lot of memory bandwidth. Another work [Sartor and Eeckhout 2012] explored separating GC threads to another socket and scaling down the frequency, revealing that when GC threads in particular are scaled down, there is an overall increase in execution time.

In this paper, we focus on minimizing the execution time of managed language applications running on a heterogeneous multicore through scheduling. If garbage collection is performed in STW mode, it is obvious that it is critical (i.e., holding up the progress of the application), and thus should be transferred to the big core, even if the heterogeneous system has limited big core resources. However, the problem is more complex with a concurrent collector that runs alongside the application, which has to coordinate during allocations and writes to references. Furthermore, the GC and application compete for core and memory resources. Of course the application's progress is most critical; however, if GC has to stop the application to finish scanning the heap, it becomes the critical path. The criticality of concurrent GC depends on how fast the application is using memory (including its allocation rate and object sizes and lifetimes), and how fast the collector is able to free up memory. We aim to design a scheduler that responds to GC criticality by receiving hints from the managed runtime, dynamically adapting the GC's share of big core cycles to achieve the best application performance.

3. CONCURRENT GC ON HETEROGENEOUS MULTICORES

Before presenting our adaptive scheduling algorithm, we first explore the behavior of concurrent GC threads on heterogeneous multicores in more detail, to further motivate the need for an improved scheduling algorithm and to indicate the potential of heterogeneity. In our first experiment, we assess the behavior of concurrent garbage collection threads running on different core types, small versus big. We use two GC threads (as mentioned in Section 2.2, this means that there are two concurrent and two STW threads); we also pin threads to cores, and set the number of cores equal to the number of threads. To assess GC's behavior on the different core types, we compare a run using eight big (out-of-order) cores for all threads to a run using six big cores for all non-GC and

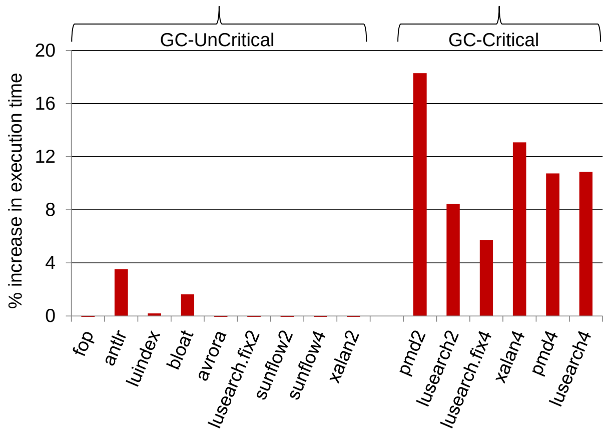


Fig. 2: Total execution time increase when concurrent GC threads are run on small versus big cores, with each other thread always pinned to its own big core. *Six multi-threaded applications are GC-critical, while the others are GC-uncritical.*

STW GC threads and two small (in-order) cores for the two concurrent GC threads (see Section 5 for more methodological details).

Some multi-threaded benchmarks exhibit GC criticality*, while other benchmarks do not.* Figure 2 shows the percentage increase in total application execution time when concurrent GC threads are run on small versus big cores, normalized to when all GC threads are on the big cores. Figure 2 shows that the execution time difference can go up to 18% for pmd2. All but one four-threaded application, and two two-threaded applications, have a large difference in execution time, which corresponds with an increase in the time spent in stop-the-world mode. On the other hand, most single-threaded, and few multi-threaded, benchmarks (antlr, bloat, fop, luindex, avrora, lusearch-fix2, sunflow2, xalan2, and sunflow4) see no execution time difference. antlr sees a small execution time change because the concurrent GC time has grown. We find that avrora and sunflow, despite having many application threads, are compute-intensive, and do not spend much time performing garbage collection. We call these nine left-most benchmarks *GC-uncritical*. The six right-most benchmarks: pmd2, lusearch2, lusearch-fix4, xalan4, pmd4, and lusearch4, have a large execution time difference when concurrent GC threads run on the small versus big cores; i.e., they exhibit *GC criticality* during execution.

Scheduling concurrent garbage collection on small cores slows down GC-critical benchmarks. The large difference in execution time for the GC-critical benchmarks when concurrent GC threads run on small cores is due to longer stop-the-world pauses. These longer pauses are due to more optional "scan" pauses, as shown on the right in Figure 1. We find that other STW phases, "roots" and "release" are relatively short on average. What increases execution time substantially is when the concurrent collection threads cannot scan and free memory in time before an application allocation request fails, and the world must be stopped for GC threads to finish scanning the heap. This is more likely to happen in multi-threaded benchmarks, where many threads are rapidly allocating memory, which increases the amount of GC work and time [Du Bois et al. 2013b]. Avoiding the critical and crippling STW "scan" phases is key to improving GC, and therefore overall application performance.

Concurrent garbage collection exploits ILP on the big core. To better understand why concurrent GC benefits from running on a big core for the GC-critical benchmarks, we present the CPI stacks [Eyerman et al. 2006] for the application threads versus the concurrent garbage collection threads in Figure 3 when running on big versus small cores.

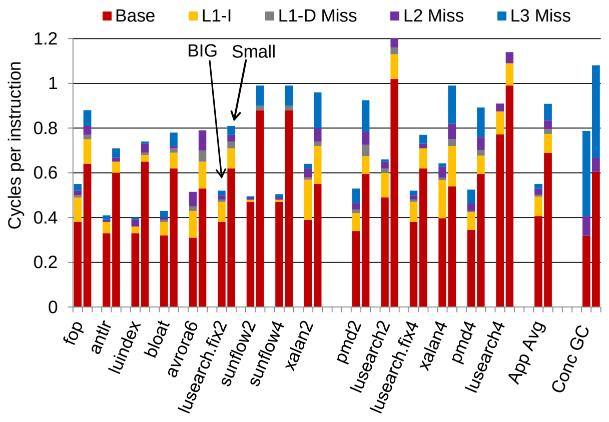


Fig. 3: The CPI stacks of only the application threads and of the concurrent garbage collector (rightmost bars) when run on big (left bar) and small (right bar) cores. *Concurrent GC threads exploit more ILP on the big core, nearly halving the CPI base component.*

To isolate application threads' behavior on the different core types, we run these threads on all big versus all small cores. Each stack shows the base component, representing committed instructions and useful work done, and the memory components, including time waiting for cache and memory accesses. The total cycles per instruction is the sum of the base and the memory components. We find that the concurrent GC threads (rightmost bars called "Conc GC") benefit substantially from running on the big out-of-order versus the small in-order core, with the benefit coming primarily from a substantial reduction in the base component. This suggests that the out-of-order core is able to exploit instruction-level parallelism (ILP) in the concurrent GC threads, hiding instruction latencies and inter-instruction register dependencies. While the collector stacks show a large memory component, larger than that of our applications, we observe there is limited memory-level parallelism (MLP), as there is little change in the memory component between the big and small core runs.

3.1. Generalizing to Different Garbage Collectors

We want to demonstrate that GC criticality is not just a function of the GC algorithm we are using. Thus, we perform experiments analyzing the behavior of Jikes RVM's best-performing production collector on both big and small cores to show that GC in general can benefit from out-of-order processing. Immix is a generational, stop-the-world collector. We run the Immix collector with two threads pinned to two separate cores on the Sniper simulator, and other experimental setup details (such as heap size) are the same as in the concurrent GC experiment. We always place application threads on out-of-order cores. Figure 4 shows the percentage increase in total execution time from running the Immix collector threads on in-order versus out-of-order cores. The benchmarks identified as GC-critical when running with the concurrent collector in the paper (on the right) also see an increase in execution time when Immix runs on the big cores: up to 15%. We also see a large execution difference for xalan2 with the stop-the-world collector. The overall conclusion is that garbage collection exhibits ILP and thus benefits from running on a big out-of-order core in a heterogeneous multicore machine.

To further show that GC criticality exists in other environmental setups as well, we perform experiments with another JVM, OpenJDK. We run the DaCapo benchmarks with OpenJDK's concurrent collector on real hardware, and use frequency scaling. The results in the appendix show that the same benchmarks that exhibit GC criticality with Jikes also show GC criticality with a different JVM.

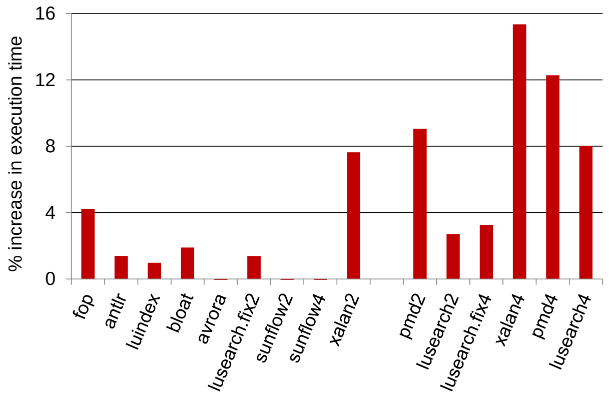


Fig. 4: Total execution time increase when STW GC threads are run on small versus big cores, with each other thread always pinned to its own big core. *The production garbage collector also benefits from out-of-order execution.*

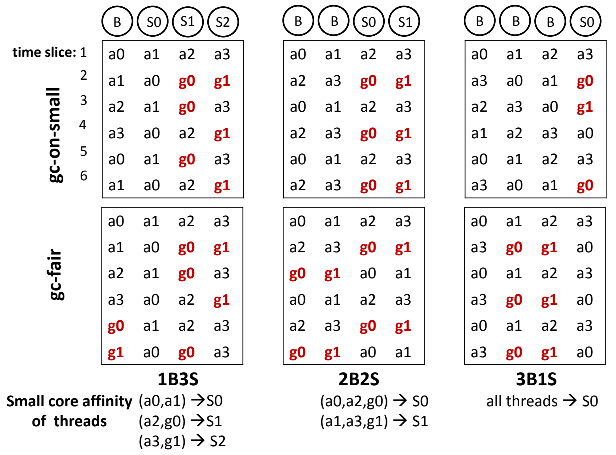
4. GC-CRITICALITY-AWARE SCHEDULING

Our adaptive scheduling algorithm measures GC criticality during run-time and dynamically adjusts the GC's priority to run on the big core(s) based on feedback about STW pauses, particularly detrimental scan pauses, incurred by the concurrent collector. The algorithm is reactive, but tries to keep GC threads on the small core(s) when GC is not critical, to let application threads use the big core cycles, while sharing big core time slices between the GC and application threads when GC is critical.

4.1. Base Schedulers

Before describing the GC-criticality-aware scheduler in more detail, we first revisit previously proposed schedulers on which we build and to which we compare. We discuss two such schedulers: one called *gc-on-small* that keeps concurrent GC threads on small core(s), and a second we call *gc-fair* that gives all threads equal time on the big core(s). The first scheduling approach, *gc-on-small*, is patterned after the recommendations of previous research to always put GC threads on small cores for better energy usage [Cao et al. 2012]. In this paper, we use this scheduler as a baseline. The second, *gc-fair*, uses the algorithm proposed in Van Craeynest et al. [2013], which was devised for native multi-threaded workloads, and was not previously evaluated for managed language workloads. This scheduler gives all runnable threads an equal percentage of time on the big core in a round-robin manner. Each time slice, the thread with the least cumulative big core time is picked to move to a big core. This implies that with four application threads, *gc-fair* would give two GC threads 33% of time slices on big cores, whereas with two application threads, it would give 50%, and with one, 66%.

The two base schedulers are graphically depicted in Figure 5. Each row of boxes shows a different scheduler, and the columns depict different four-core heterogeneous architectures. We denote the architecture of a heterogeneous multicore as mBnS, with m big cores and n small cores. We vary the number of big cores across these heterogeneous configurations: 1B3S, 2B2S, 3B1S. The contents of each box then shows which thread would be scheduled on which core, showing scheduling decisions for the first six time slices. We consider four application threads and two GC threads in this figure. These algorithms only change the scheduling of the concurrent GC threads (i.e., *g0* and *g1* from Figure 1), as we always put STW GC threads on the big core(s) because the application is no longer running. The bottom of Figure 5 also depicts that in these base schedulers, each thread has an affinity to a particular small core to exploit locality. However, the schedulers will schedule a thread waiting to run on any available idle core.



	time slice	1B3S B	1B3S S0	1B3S S1	1B3S S2	2B2S B	2B2S B	2B2S S0	2B2S S1	3B1S B	3B1S B	3B1S B	3B1S S0
gc-on-small	1	a0	a1	a2	a3	a0	a1	a2	a3	a0	a1	a2	a3
	2	a1	a0	g0	g1	a2	a3	g0	g1	a3	a0	a1	g0
	3	a2	a1	g0	a3	a0	a1	a2	a3	a2	a3	a0	g1
	4	a3	a0	a2	g1	a2	a3	g0	g1	a1	a2	a3	a0
	5	a0	a1	g0	a3	a0	a1	a2	a3	a0	a1	a2	a3
	6	a1	a0	a2	g1	a2	a3	g0	g1	a3	a0	a1	g0
gc-fair		a0	a1	a2	a3	a0	a1	a2	a3	a0	a1	a2	a3
		a1	a0	g0	g1	a2	a3	g0	g1	a3	g0	g1	a0
		a2	a1	g0	a3	g0	g1	a0	a1	a0	a1	a2	a3
		a3	a0	a2	g1	a0	a1	a2	a3	a3	g0	g1	a0
		g0	a1	a2	a3	a2	a3	g0	g1	a0	a1	a2	a3
		g1	a0	g0	a3	g0	g1	a0	a1	a3	g0	g1	a0

Small core affinity of threads

1B3S	2B2S	3B1S
(a0,a1) → S0	(a0,a2,g0) → S0	all threads → S0
(a2,g0) → S1	(a1,a3,g1) → S1	
(a3,g1) → S2		

Fig. 5: Picture depicting two schedulers across heterogeneous architectures using four application and two GC threads.

The base schedulers both have limitations. *gc-on-small* keeps the concurrent GC threads on the small core(s), which may lead to substantial performance losses for GC-critical applications. *gc-fair*, on the other hand, takes away big core cycles from the application thread(s) when scheduling GC on the big core(s), which may be detrimental for GC-uncritical applications.

4.2. GC-criticality-aware Scheduler

Developing a scheduling algorithm for concurrent GC on a heterogeneous multicore is not trivial. GC criticality is not only a function of the application and system architecture, including the number of cores, ratio of big to small cores, clock frequencies, etc. It is also a function of the GC algorithm and heap size. GC criticality is a dynamic characteristic that is based on the application's allocation speed versus the collection speed. An application becomes GC-critical if its threads progress faster, thus allocating objects faster and needing GC to collect memory faster. Thus, statically determining GC criticality for a particular application run, and choosing between *gc-on-small* and *gc-fair* is not enough. We need an adaptive GC-criticality-aware scheduling algorithm that is robust across system architectures and workload execution variations.

The fundamental principle and key insight of our adaptive scheduling algorithm is to schedule collector threads on small cores unless GC is currently critical to the application's progress; if GC is critical, we give GC threads some big core cycles, and if it remains critical, we continue to give GC more big core cycles so that it can keep up with the application and does not need to stall to clean up memory during a long stop-the-world pause. Our dynamic algorithm to schedule GC on heterogeneous cores is shown in Algorithm 1. We always start with the *gc-on-small* scheduler. We use the notion of a sampling interval (T_s) during which we profile the behavior of the garbage collector, measuring the crippling STW scan time in particular, which we aim to reduce with this algorithm. We then react to that in the next time interval, giving GC threads more big core cycles if they incurred STW scan time, and fewer cycles if there was none. Note that the mandatory STW pause (shown in Figure 1 as roots), marks the beginning of a new sampling interval.

A single sampling interval is shown in Figure 6, and each interval begins when the application encounters an STW pause. The managed runtime's memory manager communicates the beginning and end of any STW scan pause to the scheduler (i.e., the extra solid lines for threads *g2* and *g3* on the right in Figure 1). The scanning pause is only encountered when the concurrent GC could not keep up with application allocation, indicating GC criticality. During a particular sampling interval, we sum up all the STW scan pauses (T_{scan}). If T_{scan} is greater than a *noise-margin* (100μs), the

ALGORITHM 1: Our GC-criticality-aware scheduler. T_s is the sampling interval. I_{max} is the threshold for the number of consecutive intervals when GC is observed not to be critical before degrading to a *gc-on-small* scheduler.

```
input: $T_s$, $I_{max}$
initial scheduler is gc-on-small;
noise-margin = 100 micro seconds;
while true do
    wait for an STW pause;
    start a new sampling interval;
    $T_{scan} = \sum_{i=1}^{n} T_{scan(i)}$;
    end of a sampling interval;
    if scheduler is gc-on-small then
        if $T_{scan} \geq$ noise-margin then
            new scheduler is gc-boost;
        end
    end
    if scheduler is gc-boost then
        if $T_{scan} \geq$ noise-margin then
            zero-scan-intervals = 0;
            upgradeGCBoostState();
        end
        if $T_{scan} <$ noise-margin then
            degradeGCBoostState();
            zero-scan-intervals ++;
            if zero-scan-intervals = $I_{max}$ then
                new scheduler is gc-on-small;
            end
        end
    end
end
```

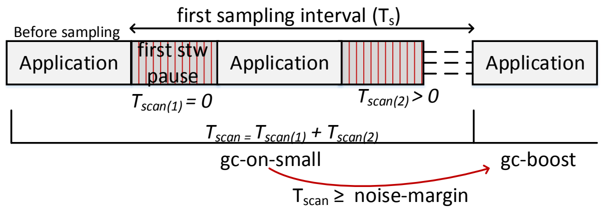


Fig. 6: A single sampling interval in our GC-criticality-aware scheduler.

scheduler switches from *gc-on-small* to *gc-boost* scheduling. Initially, *gc-boost* gives the GC threads equal priority with the application threads to run on the big core(s), effectively being the same as *gc-fair*. If, in subsequent sampling intervals, scan time continues to be significant, we *further increase* the priority of the GC threads, giving them even more time slices on the big core(s), thus implicitly slowing some application threads. If, on the other hand, in subsequent intervals scan time is zero, i.e., no GC criticality, we decrease GC's big core priority to give more time slices to application threads. If no scan time is observed for several intervals, we put GC threads back to run only on small cores. In this way, we continuously profile the garbage collector and update our scheduling policy, adapting to application phase behavior.

State	Big-core quantum of thread t, Q(t)
P0	Q(g0) = 1 ms, Q(g1) = 1 ms
P1	Q(g0) = 2 ms, Q(g1) = 1 ms
P2	Q(g0) = 2 ms, Q(g1) = 2 ms
P3	Q(g0) = 3 ms, Q(g1) = 2 ms
P4	Q(g0) = 3 ms, Q(g1) = 3 ms

Table I: GC boost states, updated/degraded by our algorithm.

Component	**Parameters**
Processor	1 socket, 4 cores, 2.66 GHz
Big core	4-issue, out-of-order, 128-entry ROB
Small core	4-issue in-order, stall-on-use
Cache hierarchy	Private L1-I/L1-D/L2, Shared L3 Capacity: 32 KB/32 KB/256 KB/16 MB Latency : 2/2/11/40 cycles Set-associativity: 4/8/8/16 64 B lines, LRU replacement
Coherence protocol	MESI
Memory controller	FR-FCFS scheduling 30.4 GB/s bandwidth to DRAM

Table II: Simulated system parameters.

We regulate GC thread priority using the states depicted in Table I. Our scheduling time slice is $1\,ms$, and thus when initially transitioning to *gc-boost* scheduling, we are in state $P0$ where each GC thread gets a $1\,ms$ time quantum on the big core in a round-robin fashion with application threads. When Algorithm 1 calls *upgradeGCBoostState* to boost GC threads' priority, our algorithm goes up one state, giving one GC thread yet another big core time slice. We allow GC threads to go up to state $P4$ which gives each GC thread $3\,ms$ on a big core when it is scheduled there. Similarly, if our algorithm discovers insignificant scan time, or that GC is not critical, it calls *degradeGCBoostState*, which decreases GC threads' priority on the big core(s) by going down one state. We also provide a counter, *zero-scan-intervals*, that is incremented every consecutive interval we see no GC criticality, and if it reaches a certain threshold, I_{max}, we switch back to the *gc-on-small* scheduler, to maintain energy efficiency.

5. EXPERIMENTAL SETUP

For evaluating GC-criticality-aware scheduling, we use the experimental setup outlined in this section.

Simulator. We perform our experiments on a simulator to evaluate scheduling algorithms across a range of potential heterogeneous architectures more easily. We use Sniper [Carlson et al. 2011] version 4.0, a parallel, high-speed and cycle-level x86 simulator for multicore systems. We use the most detailed core model in Sniper. Because Sniper is a user-level simulator, it was extended by Sartor et al. [2014] to correctly run a managed language runtime environment including dynamic compilation, and emulation of frequently used system calls. Java applications are run to completion in our experiments. For reported energy results, we use McPAT version 1.0 [Heirman et al. 2012] with a 22nm technology node.

Processor architecture. We simulate a single-ISA heterogeneous multicore processor consisting of big out-of-order and small in-order cores, as described in Table II. We set both core types to run at the same clock frequency, although we explore a small core with reduced frequency in Section 6.3. The core types also have the same cache hierarchy, i.e., each core has private L1 (32 KB) and L2 (256 KB) caches; the last-level L3 cache (16 MB) is shared among all cores. Most of our experi-

Benchmark	Heap size [MB]	Execution time (ms)
avrora6	98	1,250
luindex	44	499
fop	80	322
antlr	48	811
pmd4	98	1,150
pmd2	98	950
sunflow2	108	5,917
sunflow4	108	3,065
bloat	66	4,633
xalan2	108	1,793
lusearch.fix2	68	1,779
xalan4	108	1,136
lusearch.fix4	68	1,134
lusearch4	68	4,431
lusearch2	68	4,878

Table III: The heap sizes we use for running our benchmarks from the DaCapo suite and execution time with Sniper using a big core per thread.

ments set the total number of cores to four, and we vary the ratio of big and small cores, exploring the following configurations: 1B3S, 2B2S, 3B1S. We explore configurations with a total of six cores in Section 6.4.

JVM-scheduler communication. Our main contribution is a dynamic scheduler for heterogeneous processors that reacts to signals from the software's memory manager about GC criticality. Therefore, we modify Jikes' garbage collector to send signals to the simulator, as done by Sartor et al. [2014], using a *magic* instruction. During execution, the memory manager sends signals when STW GC threads start or stop, and in particular when they perform scanning. The thread scheduler, which is implemented in the simulator, then adapts its schedule. In all of our schedulers we use a time slice of one millisecond. The overhead of migrating threads between cores is accounted for, including restoring architecture state (we use a fixed cost of 1000 cycles), plus cache warming effects.

Java workloads. We use Jikes RVM 3.1.2 [Alpern et al. 2000] to evaluate ten Java benchmarks from the DaCapo suite [Blackburn et al. 2006] that we can get to work on our infrastructure. We use six benchmarks from the DaCapo-9.12-bach benchmark suite (avrora, luindex, lusearch, pmd, sunflow, xalan) and three benchmarks from the DaCapo 2006 benchmark suite (antlr, bloat, and fop). We also use an updated version of lusearch, called lusearch-fix (described by Yang et al. [2011]), that eliminates needless allocation. Four benchmarks, antlr, bloat, fop, and luindex, are single-threaded while the remaining benchmarks are multi-threaded. The avrora benchmark uses a fixed number (six) of application threads, but has limited parallelism [Du Bois et al. 2013b]. For the remaining multi-threaded benchmarks, we perform evaluation with two and four application threads, resulting in a total of fifteen workloads (we place the number of application threads after the benchmark's name). We vary the number of threads to explore the space because, as mentioned by Du Bois et al. [2013b], the amount of GC work and time increases as the number of application threads increase, and thus, GC can become more critical. Table III lists our benchmarks, the heap size we use in experiments (reflecting moderate, reasonable heap pressure [Sartor et al. 2014]), and their running time when using Sniper with one big core per thread.

We use replay compilation [Blackburn et al. 2008], current practice in rigorous Java performance evaluation, to eliminate non-determinism introduced by the adaptive compiler. Based on previous profiling runs, the optimization level of each method is recorded for the run with the lowest execution time. The JIT compiler then uses this optimization plan in our measured runs, picking the correct level the first time it sees each method [Huang et al. 2004]. To eliminate the perturbation of the compiler, we measure results during the second invocation, which represents application steady-state behavior. We run each application four times, and report averages in the graphs.

Concurrent collector. The concurrent garbage collector in Jikes RVM is an implementation of the mark-sweep snapshot-at-the-beginning algorithm described by Yuasa [1990]. Jikes' concurrent collector runs with n stop-the-world threads and n concurrent threads, where n is a command-line parameter. In our work, we set $n = 2$, as previous research [Du Bois et al. 2013b] shows that Jikes performs best with two GC threads, even with single-threaded benchmarks, and it does not scale well with GC thread counts above two. Because the application is not running during STW mode, in this work we always schedule the STW GC threads on the big core(s). If there is only one big core, we schedule one STW thread on the big core and leave the other on the small core because GC is a work-stealing algorithm. We use a default heap size specified in Table III, which we vary in a sensitivity study in Section 6.5. A concurrent GC cycle is triggered to begin after every 8 MB of allocation (after the previous cycle finishes).

6. EXPERIMENTAL EVALUATION

We now evaluate our GC-criticality-aware scheduler in terms of performance and energy-efficiency, across a range of heterogeneous multicore processors and configurations.

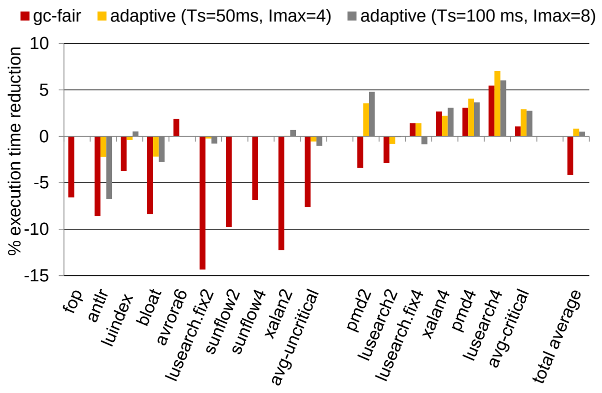
6.1. Performance

Figure 7 presents per-benchmark performance results for our adaptive, GC-criticality-aware scheduler on three heterogeneous architectures, also comparing to the execution time reduction of the *gc-fair* scheduler. In all results, we normalize to when GC is run on small cores (*gc-on-small*). The graphs present our adaptive scheduler results using different algorithm parameter configurations, with a sampling interval of 100 *ms* and I_{max} of 8, as well as an interval of 50 *ms* and I_{max} of 4. Our graphs present averages per category: for benchmarks identified as GC-uncritical, GC-critical, and then a total average across all benchmarks.

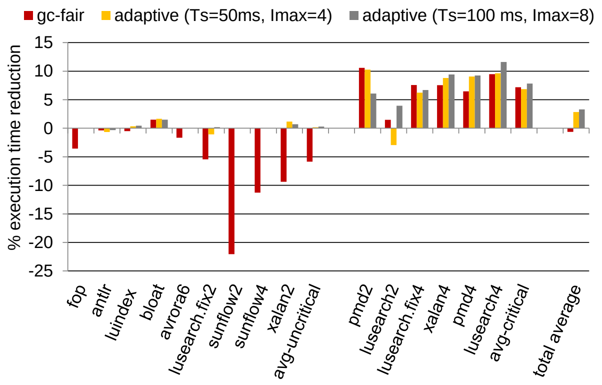
GC-criticality-aware scheduling performs well across heterogeneous architectures, greatly improving the performance of GC-critical applications over gc-on-small. Looking at individual benchmark trends in Figure 7, we see the same six benchmarks that Figure 2 identified as GC-critical as those that benefit most from our adaptive scheduling algorithm. We see a clear trend that performance gains increase as we add more big cores. For the three heterogeneous architectures, 1B3S, 2B2S and 3B3S, we observe an average performance improvement of 2.9%, 7.8%, and 16%, respectively, for the GC-critical benchmarks. The reason for this increase is that with more big cores, application threads run, and thus allocate, faster. GC becomes more critical, hence boosting the priority of GC through our GC-criticality-aware scheduler becomes more beneficial.

GC-criticality-aware scheduling improves over gc-fair *for heterogeneous multicores with limited big core resources.* On a 1B3S architecture, *gc-fair* severely degrades performance for GC-uncritical benchmarks, whereas our adaptive scheduler is on-average performance-neutral. Furthermore, for GC-critical benchmarks, our algorithm generally sees slightly higher performance improvements than *gc-fair*, with some results being similar or slightly lower due to the reactive nature of our scheduler. We see larger reductions in execution time when our algorithm responds to phase behavior about GC criticality and boosts the number of big core cycles given to GC threads over *gc-fair*.

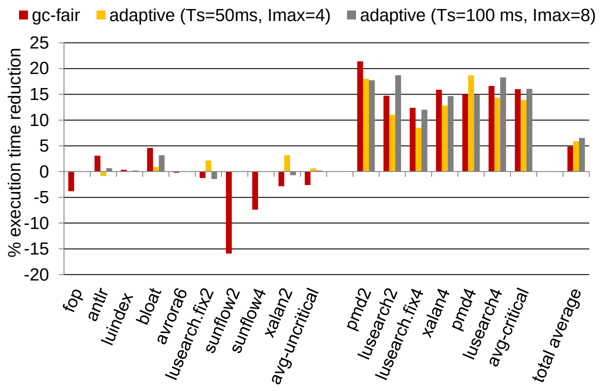
We show one such case in Figure 8 for pmd4 on 1B3S. The x-axis shows each sampling interval, and the y-axis shows the portion of that 100 *ms* interval that was measured as being stop-the-world, as well as the STW time just for scanning. In the 2nd interval, STW scan time is significant, hence the scheduler switches to *gc-boost* in the next interval. As STW scan time keeps on being significant, the GC boost state (shown at the top) is increased up to $P4$, giving more big core cycles to GC threads. GC threads continue to be critical, as they cannot scan the heap fast enough to keep up with application allocation. We see that in interval 9, GC finally becomes non-critical, but again becomes



(a) 1B3S



(b) 2B2S



(c) 3B1S

Fig. 7: Percentage execution time reduction of our adaptive and *gc-fair* schedulers compared to *gc-on-small* on three heterogeneous architectures. *Our adaptive scheduler is robust across architectures, rarely degrading performance for GC-uncritical applications, and improving performance the same or more than fair for GC-critical applications.*

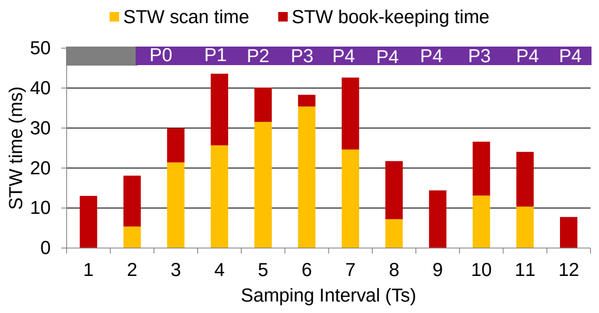


Fig. 8: STW phase behavior over time for pmd4 on 1B3S with our adaptive scheduler, $T_s = 100ms$ and $I_{max} = 8$.

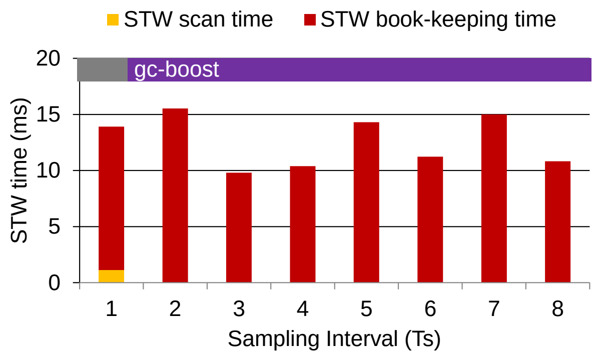


Fig. 9: STW phase behavior over time for antlr on 1B3S with our adaptive scheduler, $T_s = 100ms$ and $I_{max} = 8$.

critical in intervals 10 and 11. This case clearly illustrates the need for boosting GC priority beyond *gc-fair* while adapting to GC criticality.

GC-criticality-aware scheduling greatly reduces the negative impact of gc-fair *for the GC-uncritical applications.* On average across all benchmarks, while *gc-fair* increases total average execution time by 4% on a 1B3S architecture, our adaptive algorithm decreases execution time slightly, by 1% — a net performance improvement of 5% over *gc-fair*. We see the same trend on a 2B2S architecture, noting that our adaptive algorithm improves performance by about 3%. On a 3B1S architecture, the *gc-fair* scheduler can improve performance by about 5% over *gc-on-small*; however, it also degrades performance severely by over 15% for sunflow2. Our adaptive algorithm, on the other hand, improves performance on average by over 6%, and is more robust across applications (with no negative outliers).

Both antlr and bloat exhibit slowdowns with our adaptive algorithm on a 1B3S architecture. For antlr, using the larger sampling interval size of $T_s = 100$ and the large degradation threshold of $I_{max} = 8$ particularly hurts performance. We explain antlr's behavior in Figure 9. In antlr's 1st interval, it has a significant STW scan time, and thus our adaptive algorithm switches to the *gc-boost* scheduler. When using a large I_{max} like 8, the scheduler remains set to *gc-boost* until it sees 8 consecutive intervals where GC is not critical, meanwhile taking many big-core cycles away from the application. Because antlr is such a short-running benchmark, it never switches back to the *gc-on-small* scheduler, and thus performance is degraded because antlr is in fact GC-uncritical. Figure 7 shows that with more conservative values, $T_s = 50$, $I_{max} = 4$, antlr's performance degradation reduces.

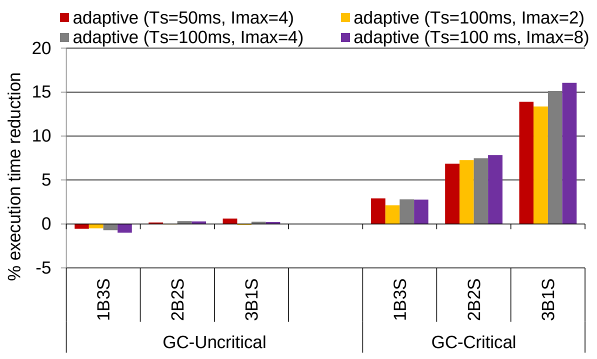


Fig. 10: Average percentage execution time reduction across benchmarks per category for different T_s and I_{max} values. *Our GC-criticality-aware scheduler is robust across its parameter settings. Small values of T_s and I_{max} are better performing with a single big core, whereas larger T_s and I_{max} are better with more big core resources.*

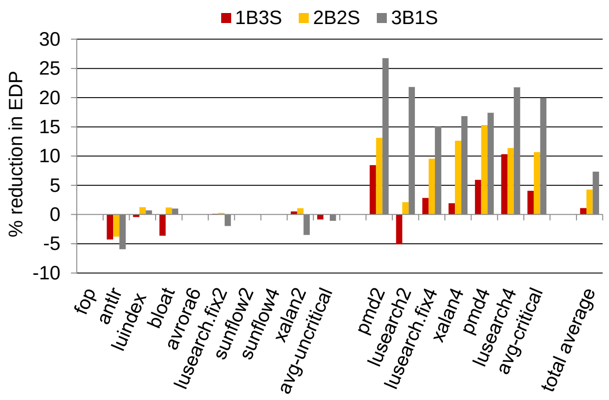


Fig. 11: Percentage reduction in energy-delay product. *By improving performance, GC-criticality-aware scheduling also improves energy efficiency over keeping GC threads on small cores.*

GC-criticality-aware scheduling is robust to its parameter settings. Figure 10 evaluates the impact of different parameters on our algorithm: with a sampling interval of 100 ms and an I_{max} of 2, 4, and 8, then setting I_{max} to 4 and using a sampling interval of 50 ms. We present averages across heterogeneous architectures per benchmark category, showing little performance variation across parameters. However, we find that, as shown in Figures 7 and 10, larger T_s and I_{max} values are not as beneficial for an architecture with only one big core. Big core cycles are more precious, i.e., taking them away from the application thread(s) hurts total performance more, especially for the short-running antlr benchmark (as shown in Figure 9). In the paper, we therefore use $T_s = 50$ and $I_{max} = 4$ for our scheduler on the 1B3S architecture to more conservatively use the big core, while for all other heterogeneous configurations, we use the default of $T_s = 100$ and $I_{max} = 8$.

6.2. Energy Efficiency

While GC-criticality-aware scheduling improves performance, it also improves energy-efficiency for GC-critical applications. We use the energy-delay product (EDP) as a metric for quantifying the energy-efficiency of GC-criticality-aware scheduler. EDP is the product of energy consumed by an application and its execution time, and thus emphasizes both energy-efficiency and performance. Figure 11 presents the reduction in energy-delay product across benchmarks and heterogeneous

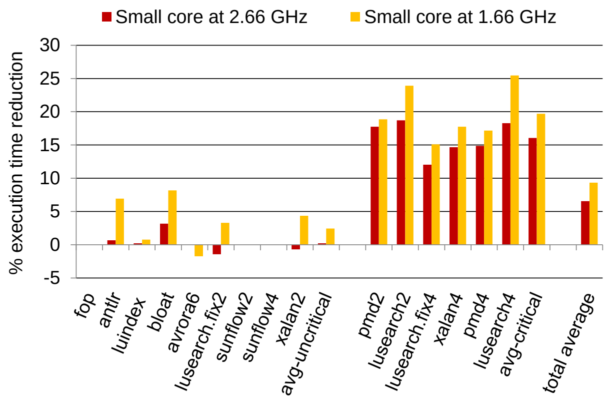


Fig. 12: Percentage execution time reduction for 3B1S when scaling down the frequency of the small core. *GC-criticality-aware scheduling improves performance more when small cores run at a lower frequency than big cores.*

architectures, relative to *gc-on-small* which was designed for energy-efficiency [Cao et al. 2012]. Our scheduler beats *gc-on-small* by a significant margin—20% on average—for the GC-critical benchmarks on the 3B1S architecture. The key insight here is that when applications will be stalled due to slow collection, giving GC time on the big core can be more energy-efficient than always leaving it on small cores. GC-uncritical benchmarks, on the other hand, are affected neutrally in terms of energy efficiency, except for antlr which sees a slight increase in EDP, for the reasons explained with Figure 9.

6.3. Scaling Small Core Frequency

While the previously-presented results are more conservative due to the big and small cores running at the same frequency, we now explore scaling down the frequency of the small core. Figure 12 presents the results of our GC-criticality-aware scheduler as we change the small core's frequency from the default of 2.66 GHz to 1.66 GHz with a 3B1S architecture. When the small core is run at a lower frequency, even some of the benchmarks categorized as GC-uncritical exhibit GC criticality in their runs. This happens because application threads run at a higher frequency compared to GC threads, which default to be on the small core and thus cannot keep up with allocation demand. With the 1.66 GHz small core, antlr, bloat, lusearch-fix2, and xalan2 see performance improvements that they did not see with 2.66 GHz. Furthermore, all six GC-critical applications see even larger execution time reductions with the small core's scaled-down frequency, leading to the GC-critical average of 20% better performance.

6.4. Larger Core Counts

Figure 13 presents performance results for heterogeneous architectures with six total cores, varying the number of big cores: 1B5S, 2B4S, 3B3S. Our scheduler uses its default parameters of $T_s = 100$ and $I_{max} = 8$. We present results for only four-threaded applications as then the number of threads and cores are equal, modeling a non-over-subscribed system. With equal numbers of threads and cores, all threads have the opportunity to progress, and thus we see less GC criticality in these configurations. However, while gains are modest, particularly for 1B5S, GC criticality does still exist for these four-threaded applications (besides sunflow4). Particularly with more big cores to divide between application and GC threads, GC-criticality-aware scheduling achieves performance improvements.

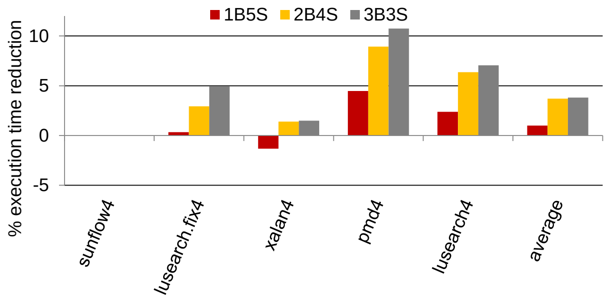


Fig. 13: Percentage execution time reduction of our adaptive scheduler for various heterogeneous architectures with six total cores and four-threaded applications. *GC criticality still exists when thread count equals core count and GC-criticality-aware scheduling still improves performance.*

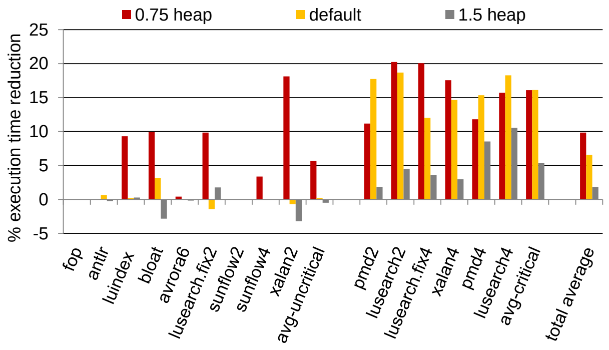


Fig. 14: Percentage execution time reduction of our adaptive scheduler with varying heap sizes (ratios of those in Table III) on 3*B*1*S*. *GC-criticality-aware scheduling performs well across heap sizes.*

6.5. Heap Size Sensitivity Study

We now show experiments varying the heap size. Figure 14 plots execution time reductions for our adaptive scheduler with a smaller and larger heap: 0.75× and 1.5× the default (in Table III) used in other experiments, for the 3B1S architecture. Our algorithm performs well across heap sizes. As when the small core's frequency is scaled down, we see that benchmarks previously labeled GC-uncritical, such as luindex, bloat, lusearch-fix2, and xalan2, exhibit GC criticality when the heap size is reduced, and have significant performance benefits from using our dynamic scheduler: up to 18%. For three GC-critical applications, we also see higher improvements with a smaller heap size, indicating that our scheduler will be more beneficial with a more constrained memory system. Naturally, when the heap grows, the application has more space into which to allocate, and thus does not encounter GC STW scan phases as much. GC-critical benchmarks, however, still realize performance improvements at the larger heap size: on average 5%. We also performed experiments using a larger trigger value (32 MB compared to our default 8 MB), which initiates concurrent collection cycles less frequently. The results show that as expected, similar to using larger heap sizes, GC criticality is reduced because collection runs concurrently with the application less. However, GC-critical benchmarks still experience scan pauses, and our GC-criticality-aware scheduler still realizes significant performance improvements.

Bottom line. Our scheduling algorithm is reactive, adapting to benchmark phase behavior, dynamically changing GC's priority on the big core(s) while the benchmark runs. For phases and executions where GC is uncritical, our algorithm keeps GC threads on the small core(s), achieving similar energy efficiency and performance as when using a *gc-on-small* scheduler. For GC-critical phases, our algorithm boosts the GC threads' priority on the big core(s), beyond *gc-fair* if needed, balancing cycles between application and GC threads. For benchmarks we identified as GC-critical, we see performance improvements of 2.9, 7.8, and 16% on heterogeneous architectures with one, two, and three big cores out of a total of four cores, respectively. We see corresponding EDP reductions of 3.5, 10.7, and 20%. Our adaptive algorithm is more robust across heterogeneous architectures than previous schedulers, and sees larger benefits when GC becomes more critical, such as with applications running with more threads, on heterogeneous multicores with more big cores, with a smaller heap size, and/or with small cores running at a lower frequency.

7. RELATED WORK

We now describe work related to scheduling multithreaded and managed workloads on (heterogeneous) multicores.

7.1. Scheduling Managed Language Workloads on Heterogeneous Multicores

A number of prior works evaluate how best to schedule managed language workloads on heterogeneous multicores, suggesting that it is best to put service threads on small(er) cores. Recent work by Cao et al. [2012] explores the opportunity to isolate service threads to small cores to optimize performance per energy. They focus on service threads in isolation, and argue that as garbage collection runs asynchronously with the application, is not on the critical path and is memory-bound, it is better to run GC on small cores. In contrast, we focus on end-to-end performance and find that GC could end up on the critical path and hurt overall application performance if always left to execute on the small core(s). We also achieve better energy efficiency in comparison with their policy.

Heil and Smith [2000] advocate for exploiting simple in-order cores to run low-priority service threads in co-designed virtual machines. Recent work [Maas et al. 2012] explores changing the garbage collection algorithm to be amenable to running on a GPU. Hu and John [2006] study core adaptation for managed applications for the purposes of energy reduction. They conclude that whereas the application benefits most from wide-issue out-of-order cores, GC threads prefer simpler cores, albeit still out-of-order but with a smaller instruction window. Our results are in line with this finding: GC benefits from periodically running on a big core to improve overall performance.

7.2. Managed Language Workloads on Multicores

Prior work explores scheduling Java workloads on modern multicore hardware to get the best performance. Sartor and Eeckhout [2012] evaluate the effect of isolating garbage collection threads to another socket, and scaling down the frequency of that socket. They conclude that slowing down the clock frequency of garbage collection degrades performance, which is in line with our findings, also noting that it degrades performance much less than when scaling down application threads.

Esmaeilzadeh et al. [2011] evaluate performance and power consumption across five generations of processors, concluding that managed language workloads are more power-hungry and exploit more parallelism than native single-threaded workloads, further motivating the relevance of our work.

Some prior work proposes hardware support for concurrent GC, adding hardware for new ISA instructions [Joao et al. 2009], to the memory subsystem [Schmidt and Nilsen 1994], to the CPU pipeline [Heil and Smith 2000], or with a completely custom design [Click 2009]. In our work, we assume 'stock' heterogeneous multicores with big and small cores.

7.3. Scheduling on Heterogeneous Multicores

The seminal work by Kumar et al. [2003, 2004] advocates single-ISA heterogeneous multicores for energy efficiency reasons. This has spurred a flurry of related work in scheduling for heterogeneous multicores; see for example [Becchi and Crowley 2006; Chen and John 2009; Ghiasi et al. 2005; Koufaty et al. 2010; Lakshminarayana et al. 2009; Li et al. 2007, 2010; Shelepov et al. 2009; Srini-

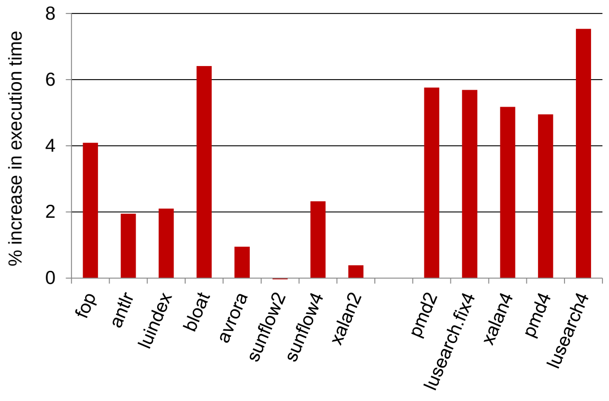


Fig. 15: Total execution time increase for OpenJDK when the GC threads, isolated on a separate socket, are run at 1.6 GHz versus 3.0 GHz. Results are with OpenJDKs concurrent generational mark-sweep collector.

vasan et al. 2011; Van Craeynest et al. 2012]. In recent work, Van Craeynest et al. [2013] propose fairness-aware scheduling for multi-threaded (native) applications. A key difference is that we acknowledge the inherent thread heterogeneity in managed language workloads, treating GC threads differently from application threads.

A number of recent works have looked into dynamically identifying and speeding up critical threads or bottlenecks [Du Bois et al. 2013a,b; Joao et al. 2012; Suleman et al. 2009]. None of these readily apply to concurrent garbage collection in managed language workloads. Our work demonstrates that providing semantic information about GC criticality from the managed runtime helps the scheduler to give big core cycles to GC threads dynamically when needed.

8. CONCLUSIONS

This paper studies how to schedule managed language applications, and concurrent garbage collection in particular, on single-ISA heterogeneous multicores. We demonstrate, contrary to prior work, that concurrent garbage collection can significantly benefit (up to 18%) from out-of-order versus in-order execution. Moreover, we find that applications exhibit GC criticality when the application allocation rate exceeds the collection rate; in this case, it is then beneficial to take big core cycles from the application to give to concurrent GC threads so that they can collect the heap faster, avoiding costly stop-the-world pauses that make GC critical. These results motivate our novel adaptive scheduling algorithm that dynamically sets the GC's priority for getting big core cycles based on GC criticality signals from the managed runtime. GC-criticality-aware scheduling improves performance by 2.9%, 7.8%, and 16% and energy-delay product (EDP) by 3.5%, 10.7% and 20% for a set of GC-critical benchmarks when using one, two, or three big cores with four total cores, respectively; while being performance-neutral for GC-uncritical applications. We demonstrate our GC-criticality-aware scheduler's robustness across core counts, big to small core ratios, heap sizes, and clock frequency settings, concluding that it is particularly beneficial as GC becomes more critical. Making schedulers aware of GC criticality leads to dynamically-optimized application performance and energy on future heterogeneous architectures.

APPENDIX: GC Criticality in OpenJDK

In this appendix, we demonstrate that some applications still exhibit GC criticality in a different environment, namely, when run on top of OpenJDK 6 HotSpot JVM using its Concurrent Mark Sweep (CMS) collector. We perform experiments on a real machine using frequency scaling. Unlike Jikes' concurrent collector, which is not generational, OpenJDK's CMS collector is generational.

We perform experiments on an eight-core machine with two Intel Xeon X5570 processors. We modify OpenJDK to pin threads that perform the GC-related tasks to a separate socket (Socket-GC). All other threads, including other service threads, run on their own socket (Socket-App). These threads are easily identified in OpenJDK by noting their ThreadType attribute (pgc-thread and cgc-thread). We run multithreaded CMS with two concurrent collection threads and two stop-the-world threads. We use the same heap size as in the rest of the paper (default). The frequency of each socket can be scaled between 1.6 GHz and 3.0 GHz. We fix the frequency of the socket running the application threads and the JVM service threads (other than the GC threads) to 3.0 GHz (Socket-App). We run benchmarks from the DaCapo suite twice, first running the Socket-GC at 3.0 GHz, and then at 1.6 GHz. Figure 15 plots the percentage increase in execution time when GC threads run at 1.6 GHz versus 3.0 GHz. Some benchmarks do not suffer from running the GC socket at 1.6 GHz, which is advantageous in terms of energy-efficiency. However, seven benchmarks observe a more than 4% increase in execution time, and up to 8%. All benchmarks (to the right) that we identified as GC-critical when running with Jikes' concurrent collector are also GC-critical with OpenJDK's concurrent collector. Our GC-criticality-aware scheduler aims to mitigate this performance loss by preemptively boosting concurrent GC threads to have more big core machine resources when they are observed to be critical.

REFERENCES

Bowen Alpern, Clement R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

Michela Becchi and Patrick Crowley. 2006. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the Conference on Computing Frontiers (CF)*. 29–40.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 25–36.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. 169–190.

Stephen M. Blackburn and Antony L. Hosking. 2004. Barriers: Friend or Foe?. In *Proceedings of the International Symposium on Memory Management (ISMM)*. 143–151.

Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32.

Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM* 51, 8 (Aug. 2008), 83–89.

Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. 2012. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *Proceedings of* *the International Symposium on Computer Architecture (ISCA)*. 225–236.

Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. 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 (SC)*. 52:1–52:12.

Jian Chen and Lizy K. John. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 927–930.

Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K. Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. 2012. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the High Performance Computer Architecture (HPCA)*. 1–8.

Cliff Click. 2009. Azul's Experiences with Hardware/Software Co-Design. Keynote at ECOOP. (2009).

Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013a. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 511–522.

Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013b. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 355–372.

Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. 2011. Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 319–332.

Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 175–184.

Soraya Ghiasi, Tom Keller, and Freeman Rawson. 2005. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings of the Conference on Computing Frontiers (CF)*. 199–210.

Peter Greenhalgh. 2011. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7: Improving Energy Efficiency in High-Performance Mobile Platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf. (Sept. 2011).

Timothy H. Heil and James E. Smith. 2000. Concurrent garbage collection using hardware-assisted profiling. In *Proceedings of the International Symposium on Memory Management (ISMM)*. 15–19.

Wim Heirman, Souradip Sarkar, Trevor E. Carlson, Ibrahim Hur, and Lieven Eeckhout. 2012. Power-Aware Multi-Core Simulation for Early Design Stage Hardware/Software Co-Optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 3–12.

Shiwen Hu and Lizy K. John. 2006. Impact of Virtual Execution Environments on Processor Energy Consumption and Hardware Adaptation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*. 100–110.

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Mutator Locality. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 69–80.

José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 418–428.

José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck Identification and Scheduling in Multithreaded Applications. In *Proceedings of the International Conference* *on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 223–234.

David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 125–138.

Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 81–92.

Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 64–75.

Nagesh B. Lakshminarayana, Jaekyu Lee, and Hyesoon Kim. 2009. Age Based Scheduling for Asymmetric Multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 25:1–25:12.

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*. 53:1–53:11.

Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of the High Performance Computer Architecture (HPCA)*. 1–12.

Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, and John Kubiatowicz. 2012. GPUs As an Opportunity for Offloading Garbage Collection. In *Proceedings of the International Symposium on Memory Management (ISMM)*. 25–36.

NVidia. 2011. Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf. (2011).

Jennifer B. Sartor and Lieven Eeckhout. 2012. Exploring Multi-threaded Java Application Performance on Multicore Hardware. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 281–296.

Jennifer B. Sartor, Wim Heirman, Steve Blackburn, Lieven Eeckhout, and McKinley McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 15–26.

William J. Schmidt and Kelvin D. Nilsen. 1994. Performance of a Hardware-assisted Real-time Garbage Collector. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 76–85.

Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009).

Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. 2011. Efficient Interaction Between OS and Architecture in Heterogeneous Platforms. *SIGOPS Oper. Syst. Rev.* 45, 1 (Feb. 2011), 62–72.

M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 253–264.

Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the international conference on Parallel architectures and compilation techniques (PACT)*. 177–188.

Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Pro-*

ceedings of the International Symposium on Computer Architecture (ISCA). 213–224.

Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 307–324.

Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181 – 198.

Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Proceeding of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 361–376.