Optimizing Soft Error Reliability Through Scheduling on Heterogeneous Multicore Processors

Ajeya Naithani, Stijn Eyerman, and Lieven Eeckhout

Abstract—Reliability to soft errors is an increasingly important issue as technology continues to shrink. In this paper, we show that applications exhibit different reliability characteristics on big, high-performance cores versus small, power-efficient cores, and that there is significant opportunity to improve system reliability through reliability-aware scheduling on heterogeneous multicore processors. We monitor the reliability characteristics of all running applications, and dynamically schedule applications to the different core types in a heterogeneous multicore to maximize system reliability. Reliability-aware scheduling improves reliability by 25.4 percent on average (and up to 60.2 percent) compared to performance-optimized scheduling on a heterogeneous multicore processor with two big cores and two small cores, while degrading performance by 6.3 percent only. We also introduce a novel system-level reliability metric for multiprogram workloads on (heterogeneous) multicores. We provide a trade-off analysis among reliability-, power- and performance-optimized scheduling, and evaluate reliability-aware scheduling under performance constraints and for unprotected L1 caches. In addition, we also extend our scheduling mechanisms to multithreaded programs. The hardware cost in support of our reliability-aware scheduler is limited to 296 bytes per core.

Index Terms—Reliability, soft errors, heterogeneous architectures

1 Introduction

As technology shrinks and operation voltage decreases, the amount of charge in a transistor's gate reduces, which increases the probability that a charged element or radiation can flip the content of a bit, a phenomenon referred to as a soft error [2], [3], [4]. A higher soft error probability implies a shorter mean time to failure, or reduced dependability. A significant body of work seeks at improving resilience to soft errors, see for example [3], [5], [6], [7], [8], [9].

To the best of our knowledge, how heterogeneous chip-multiprocessors (HCMPs) affect reliability is a largely unexplored topic. HCMPs enable high performance and high power/energy-efficiency by scheduling applications to big, high-performance cores versus small, low-power cores based on the applications' characteristics [10], [11]. Industry examples of single-ISA heterogeneous multicores include ARM's big.LITTLE [12], NVidia's Tegra [13], and Intel's QuickIA [14]. Prior work in scheduling for HCMPs focused on optimizing performance [15], energy efficiency [16], and power efficiency [17], [18]. However, no prior work has explored scheduling for reliability on HCMPs.

- *A. Naithani and L. Eeckhout are with the Department of Electronics and Information Systems, Ghent University, Ghent, East Flanders 9052, Belgium. E-mail: {ajeya.naithani, lieven.eeckhout}@ugent.be.*
- *S. Eyerman is with Intel, Kontich 2550, Belgium. This work was done while he was at Ghent University'. E-mail: stijn.eyerman@intel.com.*

Manuscript received 9 May 2017; revised 18 Sept. 2017; accepted 8 Nov. 2017. Date of publication 3 Dec. 2017; date of current version 16 May 2018. (Corresponding author: Ajeya Naithani.)
Recommended for acceptance by J. Henkel.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2779480

An HCMP features different core types, with each core type exposing different performance and soft error vulnerability characteristics. A big out-of-order core features substantially more transistors, and is therefore more vulnerable to bit flips than a small core. On the other hand, a big core executes an application faster, reducing its exposure to soft errors between launching and finishing the application. The difference in soft error vulnerability across core types and applications opens opportunities for scheduling to improve system reliability.

In this paper, we propose reliability-aware scheduling for HCMPs. The scheduler monitors reliability on either core type for all of the co-running applications, and schedules the applications to big and small cores for improved overall system reliability. The scheduler adapts to dynamic phase changes in the workload, while relying on a novel soft error vulnerability metric, called *System Soft Error Rate (SSER)*, for quantifying system reliability of multiprogram workloads on (heterogeneous) multicores. The scheduler leverages a counter architecture to track occupancy in various hardware structures. The hardware cost for the counter architecture amounts to 904 bytes per core for the baseline version; the area-optimized version requires as little as 296 bytes per core. Reliability-aware scheduling reduces system soft error rate by 32 percent on average (and up to 55.6 percent) for four-program workloads on an HCMP with two big and two small cores compared to random scheduling, while yielding similar performance. Compared to performance-optimized scheduling, soft error rate is reduced by 25.4 percent on average (and up to 60.2 percent), while degrading performance by 6.3 percent only.

This journal paper is an extension upon the previously published paper at the 2017 HPCA conference [1]. In the conference paper, we showed that our scheduler performs well across core count, number of big versus small cores, and frequency settings. In this journal paper, we explore the trade-off in reliability-, power- and performance-optimized scheduling; we demonstrate how to extend reliability-aware scheduling under performance constraints, i.e., we optimize reliability while not degrading performance by more than a predefined threshold; we evaluate reliability-aware scheduling for multi-threaded workloads and conclude that there is limited opportunity because of the homogeneous nature of data-parallel workloads; finally, we demonstrate the applicability of reliability-aware scheduling even in the case where L1 caches are not protected and scheduling needs to take into account the vulnerability to soft errors in the L1 caches.

Overall, we make the following contributions in this work:

- We analyze the difference in reliability characteristics between big and small cores.
- We show the potential for optimizing reliability through scheduling on HCMPs.
- We define a novel metric, System Soft Error Rate, for assessing reliability to soft errors for multiprogram workloads on (heterogeneous) multicores.
- We propose a dynamic online reliability-aware scheduler to optimize reliability in HCMPs (under performance constraints).
- We experimentally evaluate reliability-aware scheduling and show that our scheduler is able to significantly reduce vulnerability to soft errors.

The remainder of this paper is organized as follows. Section 2 analyzes the reliability characteristics in an HCMP, and shows that there is significant potential for reliability-aware scheduling. In Section 3, we propose the SSER metric for quantifying the soft error rate of multiprogram workloads. In Section 4, we then describe our reliability-aware scheduler. After detailing our experimental setup in Section 5, we evaluate and analyze our proposed scheduler in Section 6. We explore the trade-offs between performance-, power- and reliability-optimized scheduling in Section 7. We further evaluate reliability-aware scheduling under performance constraints (Section 8); we evaluate reliability-aware scheduling using multi-threaded workloads (Section 9); and we incorporate L1 cache vulnerability in our scheduler (Section 10). Finally, we describe related work (Section 11) and conclude (Section 12).

2 Motivation

We first analyze the difference in vulnerability to soft errors across core types, and then show the potential for reliability-aware scheduling using an offline oracle approach.

2.1 Terminology

Before doing so, we first introduce some terminology. An *ACE bit* (architecturally correct execution) is a bit in the processor that will cause an error during program execution when flipped, affecting user-visible state (program crash or wrong output). We assume each bit in the processor pipeline holding state of a correct-path and non-nop instruction to be ACE; i.e., all bits in the issue queue, load/store queue, reorder buffer, physical register file, and functional unit holding state of a correct-path, non-nop instruction are considered ACE. Structures that improve performance but do not affect functional correctness (e.g., a branch predictor) do not contain any ACE bits. *ACE bit count (ABC)* is defined as the total number of ACE bits over the entire execution of a program.

The *architectural vulnerability factor (AVF)* [3] is the fraction of ACE bits to the total number of bits in a structure, core or the whole processor. AVF is application-dependent, as some applications occupy more or fewer entries in the core structures, and/or have more or fewer wrong-path instructions. *Soft error rate (SER)* is the average number of errors (on ACE bits) that occur per unit of time, e.g., 0.01 errors per day, and is the reciprocal of the mean time to failure (MTTF), e.g., 100 days. *Intrinsic fault rate (IFR)* is the probability for a single one-bit error per second, or, in other words, the average number of errors per unit of time in a single one-bit cell, e.g., 10^{-6} per day; IFR depends on the technology and the environment. As such, SER can be calculated as the number of ACE bits per unit of time times IFR. Assuming IFR is constant, ABC is therefore proportional to SER.

Formally, ABC, AVF and SER of a program running on a processor core with N bits is defined as

$$ABC = \sum_{i=1}^{N} (ACE\ cycles\ for\ bit\ i) \tag{1}$$

$$AVF = \frac{ABC}{N \times Total\ cycles} \tag{2}$$

$$SER = \frac{ABC}{Total\ cycles} \times IFR. \tag{3}$$

2.2 Reliability versus Core Type

It is commonly known that different core types in a heterogeneous multicore processor exhibit different performance and power characteristics. However, different core types also exhibit differences in reliability.

There are basically three contributors to the reliability of an application running on a core:

- The size of the structures in the core that hold architecture state and are required to guarantee functional correctness. These include the register file, functional units, issue queue, reorder buffer (for an out-of-order processor), etc. The larger these structures are, the higher the probability for an error in those structures. This first contributor is thus determined by the design itself.
- The fraction of the architecturally relevant structures that an application occupies, i.e., AVF. Some applications occupy only a small fraction of these structures, or have a lot of non-architecturally relevant instructions (nops and wrong-path instructions). The smaller the occupied fraction is, the smaller the error probability. This second contributor depends on the workload.
- The performance of the application on that core type. If an application executes faster, it will finish sooner, and therefore it will be less vulnerable to errors.

Now consider a big out-of-order core and a small in-order core in an HCMP. Obviously, the big core has larger

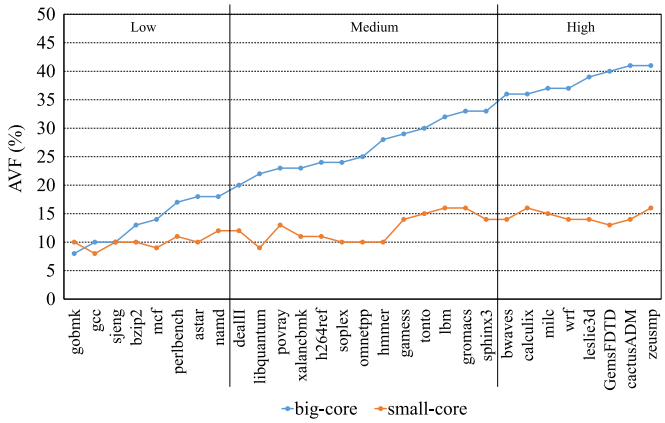

Fig. 1. AVF for the SPEC CPU2006 benchmarks on a big out-of-order and a small in-order cores. The benchmarks are sorted by their AVF on the big core.

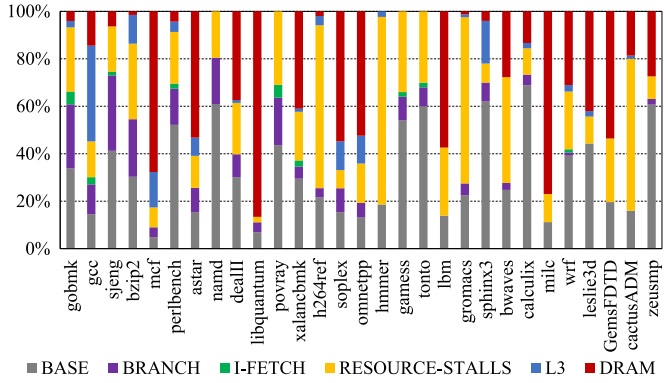

Fig. 2. Normalized CPI stacks for the SPEC CPU2006 benchmarks on a big out-of-order core.

structures than the small core. As a result, a big core is likely to expose more vulnerable state than a small core. However, the degree of vulnerability also depends on structure occupancy which is a function of the application and its performance.

2.3 Application Sensitivity

Applications exhibit varying degrees of sensitivity to soft error vulnerability. AVF is an insightful metric to understand an application's vulnerability to soft errors. Fig. 1 shows AVF for the SPEC CPU2006 benchmarks on a big out-of-order core as well as a small in-order cores. (See Section 5 for details regarding our experimental setup.) The benchmarks are sorted by their AVF on the big core. AVF accounts for all the ACE bits in the processor during the entire execution. In particular, if an ACE instruction occupies 64 bits in the reorder buffer (ROB) for 16 cycles, this amounts to 1024 ACE bits. This way of measuring incorporates structure size, occupancy and execution time. As expected, AVF is higher for the big out-of-order core compared to the small in-order core; this is because a big core holds more architecture state. Note however that in spite of the fact that AVF is higher on the small core than the big core for the left-most benchmark, `gobmk`, it is still less vulnerable to soft errors on the small core because of the smaller structure size, i.e., N is smaller.

The applications appearing on the right-hand side of the graph are most sensitive to reliability-aware scheduling, i.e., when scheduled on the big core, AVF (and thus SER) increases significantly compared to running on the small core. Applications appearing on the left-hand size are less sensitive, i.e., the increase in SER on the big core is not as high, and thus if given the choice, scheduling these applications on a big core rather than a small core will not increase overall system soft error rate as much. Fig. 1 classifies the benchmarks into three categories based on their big-core AVF: high, medium and low. We will use this classification for analyzing the performance of our reliability-aware scheduler across workload types in the evaluation section.

It is interesting to relate the AVF graph to the normalized CPI (cycles per instruction) stacks shown in Fig. 2. A CPI stack quantifies the fraction of cycles spent doing useful work (i.e., the base component) plus a number of adders or components to represent 'lost' cycles because of resource stalls, branch mispredictions, instruction cache misses, last-level cache (LLC) misses and main memory accesses. Note that the benchmarks are ordered the same way as in Fig. 1. The benchmarks on the left-hand side exhibit low AVF primarily because of their relatively high front-end miss components. Front-end miss events, such as branch mispredictions and instruction cache misses, cause the pipeline to be drained and hence there is relatively little vulnerable state in the processor. The benchmarks on the right-hand side on the other hand have a high AVF because they exhibit high occupancy in various back-end structures of the pipeline for a variety of reasons. Some benchmarks (e.g., `milc`) are memory-intensive: a load operation accessing main memory typically blocks the head of the reorder buffer, which causes the ROB to fill up, and which leads to significant ACE state while servicing the memory operation. Other high-AVF benchmarks (e.g., `zeusmp`) are compute-intensive: high IPC and high MLP is achieved by having high occupancy in various back-end queues. Yet other benchmarks experience resource stalls in the back-end structures because of L1 data cache misses, L2 cache misses, limited ILP (i.e., chains of dependent instructions) which cause the ROB and issue queues to fill up with instructions. Note that there are a number of memory-intensive benchmarks (e.g., `mcf` and `libquantum`) that exhibit low AVF. This is because these benchmarks suffer from branch mispredictions which lead to a large number of un-ACE wrong-path instructions in the ROB underneath memory accesses.

The take-away message from this analysis is that there exists no simple workload characteristic (e.g., compute-intensive versus memory-intensive) to determine how sensitive a workload is with respect to reliability. Instead, it depends on how AVF-intensive an application is, which is a result of complex interactions among various workload characteristics and the underlying microarchitecture. This suggests that reliability-aware scheduling needs a dynamic mechanism to monitor an application's reliability on either core type in a heterogeneous multicore and adjust the schedule accordingly.

2.4 Oracle Reliability-Aware Scheduling

To quantify the potential of reliability-aware scheduling, we perform the following experiment. We simulate each application on both core types in isolation, and record performance and SER. We then consider all combinations of four applications on a heterogeneous multicore processor with two big and two small cores. Of the six possible schedules, we select the one with the highest performance (expressed in

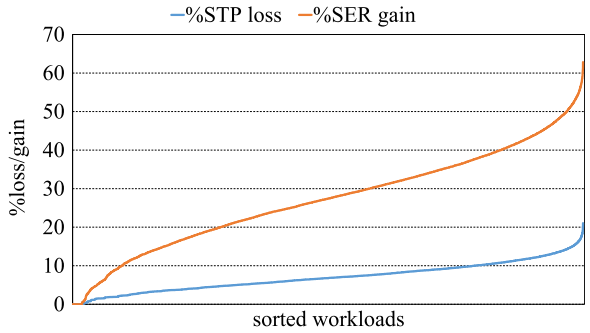

Fig. 3. Percentage STP loss and SER gain for an oracle reliability-optimized scheduler relative to a performance-optimized scheduler for four-program workloads on an HCMP with two big cores and two small cores.

system throughput (STP) [19]), and the one with the lowest total SER. (See the next section for the metric we use to quantify SER for a multiprogram workload.) We assume no interference in shared resources, and consider the performance and SER numbers from the isolated experiments. This leads to an oracle offline schedule. Fig. 3 shows SER reduction and performance loss for the SER-optimized schedule normalized to the performance-optimized schedule. Clearly, the reduction in SER is much higher than the loss in performance, resulting in an average 27.2 percent reduction in SER (and up to 62.8 percent) while degrading performance by 7 percent on average. This result demonstrates the significant potential and motivates our study on reliability-aware scheduling for heterogeneous multicore processors.

3 Reliability Metric for Multiprogram Workloads

Reliability is commonly quantified using soft error rate, i.e., the number of errors per unit of time. This works fine for single-program workloads, but falls short for multiprogram workloads, as we will explain in this section; we then subsequently propose a novel system-level reliability metric for multiprogram workloads on (heterogeneous) multicores.

With T as the total execution time (total cycles in Equation (3), let us recap the definition of soft error rate for single-program workloads

$$SER = \frac{ABC}{T} \times IFR. \tag{4}$$

In other words, SER computes the number of ACE bits per unit of time multiplied by the intrinsic fault rate. As long as we measure SER for a single-program workload by running (a well-defined section of) the workload to completion, we can safely evaluate reliability using SER because the unit of work is constant.

3.1 System Soft Error Rate

SER breaks down for multiprogram workloads. We cannot simply add up SER numbers for each of the applications in a multiprogram workload because some applications are inherently more vulnerable to soft errors than others—adding raw SER numbers would give too much weight to fast running applications and too little weight to slow running applications. This is similar to performance metrics for multiprogram workloads, i.e., adding plain IPC numbers gives more weight to high-IPC applications. The fundamental problem here is that SER does not take into account the impact of performance on the error rate: lower performance makes the application run longer, increasing the probability for an error during its execution.

The solution is to weight per-application SER with the slowdown incurred because of multiprogram execution. Application slowdown is defined as the execution time of an application on the (heterogeneous) multicore divided by its execution time on a reference machine (e.g., an isolated big core). A slowdown of 1 means that the application executes equally fast as on the reference machine; a slowdown of 2 means that the application takes twice as long under multiprogram execution compared to isolated execution. We then define *weighted SER (wSER)* of an application in a multiprogram workload as follows:

$$wSER = \frac{ABC}{T} \times \frac{T}{T_{ref}} \times IFR = \frac{ABC}{T_{ref}} \times IFR, \tag{5}$$

with ABC and T the ABC and execution time of the application in the multiprogram workload, respectively; and T_{ref} the execution time of the application on an isolated reference core (e.g., a big core in a heterogeneous multicore). In other words, wSER weights the application's SER during multiprogram execution with its slowdown compared to isolated execution. This is to account for the fact that if the application runs longer during multiprogram execution (which is what you would expect because of interference in shared resources), it gets exposed to soft errors for a longer time.

Summing the weighted SER values for the individual applications in a multiprogram workload then yields our novel *system soft error rate* metric

$$SSER = \sum_{i=1}^{n} wSER_i = \sum_{i=1}^{n} \frac{ABC_i}{T_{i,ref}} \times IFR, \tag{6}$$

which quantifies the total weighted SER across all the applications in the multiprogram workload. SSER gives bigger weights to slow-running applications in the multiprogram workload mix, and smaller weights to fast-running applications. This is to account for the fact that slow-running applications will be exposed to soft errors for a longer time, hence we scale their per-application SER proportionally with their relative slowdown.

3.2 Illustrative Examples

We now illustrate the intuitive and system-level meaning of SSER using a couple examples, see also Table 1. Consider a homogeneous multicore with two big cores, and assume that the two co-running applications do not interfere with each other, i.e., they both run equally fast on the homogeneous multicore compared to isolated core execution, see example (a) in Table 1. Assume further that per-application SER is not affected by multiprogram execution. SSER equals 2 in this case, which makes perfect sense: the system's vulnerability is twice as high on the homogeneous multicore compared to isolated execution because we now have two co-running applications.

Assume now that one application slows down by a factor of 2 (e.g., because of hardware interference) and the other application is not affected at all, see example (b) in Table 1. In this case, SSER equals 3, i.e., a weighted SER of 1 for the

TABLE 1
Examples Illustrating the SSER Metric

(a) homogeneous multicore: SSER = 2			
	SER	*slowdown*	*wSER*
benchmark A on big	1	1	1
benchmark B on big	1	1	1
(b) homogeneous multicore: SSER = 3			
	SER	*slowdown*	*wSER*
benchmark A on big	1	2	2
benchmark B on big	1	1	1
(c) heterogeneous multicore: SSER = 1.5			
	SER	*slowdown*	*wSER*
benchmark A on small	1/8	4	0.5
benchmark B on big	1	1	1

application that does not slow down, plus a weighted SER of 2 for the application that slows down by a factor two. This makes intuitive sense because it takes two times as long for the slow application to get the same amount of work done, and therefore the slow application is twice as vulnerable.

Consider now a heterogeneous multicore, see example (c) in Table 1. Assume that the application that runs on the small core experiences a slowdown of 4 while its SER reduces by a factor of 8 compared to running on the big core (we expect lower SER on the small core because it holds less state than on the big core). As a result, its weighted SER equals 0.5, i.e., the application is slowed down by a factor of 4 but it is 8 times less vulnerable to soft errors per unit of time, hence it is only half as vulnerable for getting the work done. SSER thus equals 1.5. Note that SSER in example (c) is smaller than for the homogeneous multicore examples (a) and (b); this is due to the fact that even though the benchmark running on the small core slows down substantially, it exposes far fewer ACE bits, which leads to a net reduction in overall system vulnerability.

4 Reliability-Aware Scheduling

Having demonstrated the potential for reliability-aware scheduling and having derived the SSER metric for quantifying system-level reliability, we now describe our sampling-based reliability-aware scheduler for heterogeneous multicores. We assume that we can measure the performance of each application on each core (e.g., the number of instructions executed during the last scheduler quantum), and the number of ACE bits in each structure (i.e., ACE bit count or ABC over the past quantum), which we both need to compute SSER. We quantify the hardware overhead for measuring ABC later in this section; we start by explaining the scheduling algorithm.

4.1 Scheduling Algorithm

The scheduler starts with an initial sampling phase to collect performance and ABC information for each application on either core type. If the number of big cores equals the number of small cores, this requires two sampling quanta: we first put one half of the applications on a big core and put the other half on a small core in the first sampling quantum, and we invert this schedule in the next sampling quantum, i.e., the applications running on a big core are moved to a small core, and vice versa. If the number of big cores is not equal to the number of small cores, e.g., 1 big core and 3 small cores, more quanta are needed to sample each application on each core type (4 sampling quanta in this example). After this initial sampling phase, the scheduler follows the algorithm described in Algorithm 1.

Algorithm 1. Sampling-Based Reliability-Aware Scheduler. (n is the Number of Applications.)

```
1: sampleRequired = false
2: quantumNumber = 0
3: lastSampledAt = 0
4: while true do
5:   if quantumNumber ≤ 2
       or (quantumNumber − lastSampledAt) == 10 then
6:     sampleRequired = true
7:   end if
8:   if sampleRequired == true then
9:     startSamplingPhase()
10:    lastSampledAt = quantumNumber
11:    sampleRequired = false
12:    continue
13:  end if
14:  for i = 1 to n do
15:    reduction[i] = getWeightedSERReduction(i)
16:    coreAssigned[i] = false
17:    for j = i + 1 to n do
18:      maxReduction[i,j]= 0
19:    end for
20:  end for
21:  for i = 1 to n do
22:    for j = i + 1 to n do
23:      if coreType[i] == coreType[j] then
24:        continue
25:      endif
26:      if(reduction[i] − reduction[j]) >
         maxReduction[i,j]then
27:        maxReduction[i,j] =
           reduction[i] - reduction[j]
28:      endif
29:    end for
30:  end for
     {sortedReductions contains an array of n
     maxReductions[i,j] in their decreasing order}
31:  sortedReductions[n] = sortMaxReductions()
32:  for k = 1 to n do
33:    currReduction[i,j] = sortedReductions[k]
34:    if currReduction[i,j] > 0
       andcoreAssigned[i] == false
       andcoreAssigned[j] == false then
35:      switchCoreTypes(i,j)
36:      coreAssigned[i]= true
37:      coreAssigned[j]= true
38:    endif
39:  end for
40:  for i = 1 to n do
41:    ABC[i] = getCurrentQuantumABC(i)
42:    IPC[i] = getCurrentQuantumIPC(i)
43:  end for
44:  quantumNumber++
45: end while
```

The algorithm first verifies whether the sampled data is recent. If an application has run for 10 consecutive scheduling quanta on the same core type, a sampling phase is triggered: the application is scheduled on the other core type by swapping it (during a short sampling quantum) with the application that is running for the most consecutive quanta on the other core type. By doing so, the scheduler ensures that the sample data is up-to-date, adapting to potential phase changes.

If all applications have recently sampled data for both core types, the scheduler calculates the weighted SER (wSER) for each application if we were to schedule them on the other core type than they are currently scheduled on. It then selects the application with the highest wSER reduction and the application with the smallest wSER increase, and checks whether swapping the two applications leads to a net overall SSER reduction. If so, the applications are swapped, and the next couple is checked. If no global SSER reduction can be obtained, the current schedule is maintained for the next quantum. After finishing a quantum, the sample data is automatically updated.

We need to sample both performance and ABC, because the SSER metric needs both. Sampling ABC requires hardware support to compute occupancy in all relevant processor structures, as we will describe in the next section. Sampling performance can be done by counting the number of instructions executed per quantum—we sample at fixed time quanta (1 ms in our setup). This involves a basic performance counter that is implemented in most recent processors. To compute an application's slowdown, we take the big core as the reference core. Because we have no reference performance data of an isolated big core execution, we assume that the sampled big core performance is a good proxy for reference core performance. Note that the sampled value is subject to interference in the shared resources (e.g., shared cache and memory) because other programs are co-running while sampling.

It is important for a sampling-based scheduler to limit sampling overhead. On the other hand, we need to sample for a sufficiently long period of time to obtain stable sampling information. This is why we make a distinction between a sampling quantum and a scheduler quantum. We set the scheduler quantum to 1 ms in all of our experiments, and the sampling quantum to one tenth the scheduler quantum or 0.1 ms. All results in the evaluation section include sampling overhead.

Fig. 4 illustrates how our reliability-aware scheduler reacts to time-varying execution behavior; each dot represents ABC per 1 ms. The left graph shows ABC over time for `calculix` and `povray` when executed in isolation on a big core; the right graph shows ABC when executed concurrently on an HCMP with one big and one small core. When run in isolation, `povray` experiences almost constant ABC; `calculix` on the other hand experiences a big drop in ABC towards the end of its execution. When co-executed on the HCMP, `calculix` is scheduled on the small core initially due to its high big-core ABC compared to `povray`. Upon the phase change in `calculix`, the scheduler responds by migrating the two applications. The multi-program workload case also illustrates sampling overhead: sampling is initiated once every 10 scheduler quanta for one tenth of the quantum, so we sample one percent of the time. Sampling incurs the drops and spikes in the ABC curves for `povray` and `calculix`, respectively. (Note that the curves include data points for the scheduling quanta *and* the sampling quanta, i.e., ten scheduling quanta of 1 ms each followed by a sampling quantum of 0.1 ms.)

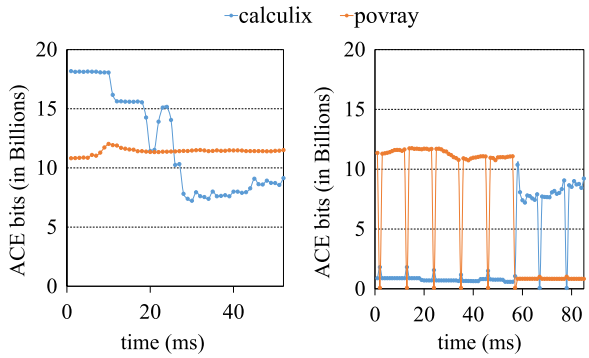

Fig. 4. ABC over time for `calculix` and `povray` when executed in isolation on a big core (on the left) and as a two-program workload on one big core and one small core under reliability-aware scheduling (on the right).

4.2 Hardware Overhead

As mentioned in the previous section, computing ABC in support of our reliability-aware scheduler requires hardware support. For an out-of-order core, we need counters for the five major structures, including the ROB, issue queue, load/store queue, register file and functional units. Furthermore, we also need to factor out wrong-path and nop instructions. We propose the following hardware additions. Per ROB entry, we keep two extra counters: one for recording the dispatch time of an instruction (i.e., the time it is inserted into the ROB), and one for recording the issue time (i.e., the time the instruction starts executing). These counters should be large enough to cover the maximum number of cycles an instruction resides in the ROB; we set the size of the counter to be 12 bits (maximum of 4,096 cycles). At the time the instruction commits—which ensures that it is a correct-path instruction—we can deduce the time this instruction spent in each of the architecturally relevant structures:

- The time spent in the ROB is the commit time minus the dispatch time.
- The time spent in the issue queue is the issue time minus the dispatch time.
- For a load or store instruction, the time spent in the load/store queue is the commit time minus the dispatch time—we model an architecture where load/store queue entries are allocated at dispatch time.
- The time the physical output register of an instruction is ACE is the commit time minus the finish time (which is the issue time plus its execution latency). Note that all architectural registers are ACE all of the time.
- The time spent in a functional unit is the functional unit's execution latency.

At the commit stage, where we keep one counter for each of the five structures, we add the per-instruction occupancy in each of the five structures to the respective overall counters.

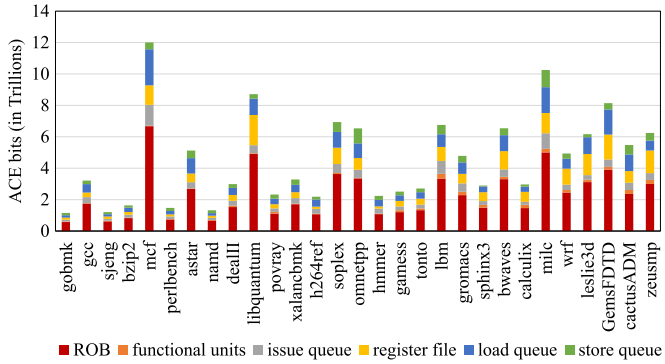

Fig. 5. ABC stacks for the out-of-order core.

By doing so, the counters keep track of the accumulated occupancy in the respective structures. At the end of a quantum, total ABC is calculated as the accumulated occupancy times the number of bits per entry—the multiplication is done by the scheduler in software.

The total hardware overhead amounts to:

- Two 12-bit counters per ROB entry, which amounts to 3,072 bits for a 128-entry ROB.
- One 32-bit counter per profiled structure, which amounts to 160 bits for 5 counters (with one counter per structure). 32 bits is sufficient for the quantum size in our setup (2.6 million cycles at 2.6 GHz, and at most 128 entries per structure).
- Additional functional units for calculating occupancy and adding them to the counter. We need 5 adders per instruction in the data path (one per structure), and since up to 4 instructions can commit per cycle, this requires 20 adders in total.

Total hardware overhead thus equals 3,232 bits plus 20 adders. Extrapolation from [20] suggests that a 32-bit adder consumes about 1,200 transistors. One SRAM cell contains 6 transistors, so a rough equivalence relation is 200 SRAM bits for one 32-bit adder. So, in total the hardware overhead of this baseline implementation equals 7,232 bits or 904 bytes.

To reduce the hardware overhead for the big core, the scheduler can use ACE bit information of the ROB only. We choose the ROB, because it is a central structure, containing a lot of useful state, and all other structures contain a subset of the instructions in the ROB. This is confirmed by the ACE bit count stacks shown in Fig. 5 for the one-billion instruction workloads considered in this study. ABC stacks represent the breakdown of the total occupancy of a core in its microarchitecture structures. ROB ABC correlates very well with overall core ABC (correlation coefficient of 0.99) and contributes to almost half of the total occupancy of the core across all benchmarks. In other words, ROB ABC can serve as a proxy for the overall core ABC, which allows for correct scheduling decisions to be made using relative ABC numbers across applications. (When using ROB-ABC instead of core-ABC in our final scheduler, we find it to be within 0.7 percent for four-program workloads running on 2 big and 2 small cores. Therefore, this is a worthwhile optimization.) For this implementation, we only need the dispatch time per ROB entry (12 bits times 128 entries equals 1,536 bits), one ROB ACE counter (32 bit) and 4 adders, resulting in a total of 2,368 bit equivalents or 296 bytes in total for this area-optimized implementation.

TABLE 2
Big and Small Core Configurations

	Big core	*Small core*
Frequency	2.66 GHz	2.66 GHz
Type	out-of-order	in-order
ROB size	128, 76 bit/entry	-
Issue queue size	64, 32 bit/entry	4, 32 bit/entry
Load queue size	64, 80 bit/entry	-
Store queue size	64, 144 bit/entry	10, 144 bit/entry
Pipeline width	4	2
Pipeline depth	8 stages (front-end only)	5 stages 2×76 bit/stage
Functional units	3 int add (1 cyc)	2 int add (1 cyc)
	1 int mult (3 cyc)	1 int mult (3 cyc)
	1 int div (18 cyc)	1 int div (18 cyc)
	1 fp add (3 cyc)	1 fp add (3 cyc)
	1 fp mult (5 cyc)	1 fp mult (5 cyc)
	1 fp div (6 cyc)	1 fp div (6 cyc)
Register file	120 int (64 bit)	16 int (64 bit)
	96 fp (128 bit)	16 fp (128 bit)
L1 I-cache	4-way 32 KB, 2 cyc	4-way 32 KB, 2 cyc
L1 D-cache	8-way 32 KB, 4 cyc	8-way 32 KB, 4 cyc
Private L2 cache	8-way 256 KB, 8 cyc	8-way 256 KB, 8 cyc
Shared L3 cache	16-way 8 MB, 30 cyc	
Memory	BW 25.6 GB/s, lat 45 ns	

For the small in-order core, we only keep track of the fetch time. Because all instructions need to go through all stages, and each stage has a similar buffer for each instruction, we can calculate the time between fetch and writeback of each instruction as a way to account for the number of ACE bits in the pipeline buffers. In addition, we add the functional unit ACE bits by multiplying the latency of the operation by the size of the functional unit. This requires 10 fetch time counters (5 stages times 2 instructions per stage) at 10 bits per counter (the time an instruction spends in the in-order core is usually less than in an out-of-order core), and one 32-bit total ACE counter. This amounts to 132 bits and two adders in total, resulting in 532 bit equivalents or 67 bytes.

5 Experimental Setup

Because there is no way of evaluating architectural vulnerability on real hardware, we evaluate our scheduler through simulation. We use Sniper 6.0 [21], a parallel, high-speed, and cycle-level x86 simulator for multicore systems that has been validated against real hardware; we assume the most detailed simulation model available in Sniper. We augment Sniper with ACE bit counters to count the number of ACE bits in the different structures. For the big out-of-order core, we count ACE bits in the ROB, issue queue, load/store queue, register file and functional units. Similarly, for the small in-order core, we count ACE bits in the fetch, decode, register read, execute and write-back stages. Nops and wrong-path instructions are assumed to be non-ACE. Table 2 shows the configurations of the big out-of-order and the small in-order core types, as well as the bit counts per entry in each structure (taken from Nair et al. [22]). Note that we assume the same cache hierarchy for the small and big cores; however, in Section 10 where we focus on L1 cache vulnerability, we will vary the cache size of the small core to study the sensitivity to the cache hierarchy.

The overhead for saving and restoring microarchitectural state to support core migration plus the overhead of weighted speedup/SER calculation is conservatively modeled as 20 μs. The impact of cache warming (including cache-to-cache transfer latency) is modeled faithfully in the simulator. The overall impact of the different overheads on system throughput is less than 0.5 percent for both the performance- and reliability-optimized schedulers.

We create multiprogram workloads from the SPEC CPU2006 benchmarks. We construct 1 billion instruction SimPoints [23] for each benchmark. We categorize benchmarks into three groups, based on their big-core AVF, see also Fig. 1. The eight benchmarks with the highest AVF are classified in the high sensitivity group (H); the eight benchmarks with the lowest AVF are classified as low sensitivity (L); and the 13 remaining benchmarks have medium sensitivity (M). For the two-program combinations, we make 6 categories of mixes: HH, HM, HL, MM, ML and LL. We randomly generate 6 workloads in each category, while making sure that each benchmark occurs at least once; this results in 36 evaluated workloads. For the four-program combinations, we take the same 6 mix categories by doubling the benchmark categories: HHHH, HHMM, HHLL, MMMM, MMLL and LLLL, and again generate 6 workloads in each category. We do not duplicate individual benchmarks, i.e., HHHH contains four different benchmarks. We do another doubling round for the eight-program combinations.

We evaluate the four-program workloads on a symmetric HCMP configuration consisting of 2 big and 2 small cores (2B2S). The standard quantum time is 1 ms. For each experiment, the longest running application executes its full 1 billion instruction SimPoint, and the faster running applications are restarted until the end of the experiment. For the applications that restart, we record performance and wSER across all repetitions of that application. The reason is that the longer running application could enter a new phase near the end of its execution, causing the schedule to change, which in turn impacts the other applications. Taking results from the first execution only for the repeating applications would not cover these changes in the schedule.

Note that in this work we assume a fixed DRAM access latency. We evaluate the impact of this assumption on the results reported in this paper by also considering a variable latency DRAM model that includes different banks and ranks, an open-page policy, and different latencies for accessing an open versus closed page. We notice that the variable-latency DRAM model leads to significant differences in absolute IPC and ABC for the individual benchmarks. However, there was no significant difference in terms of the overall reliability and performance for the evaluated schedulers when executing multiprogram workloads on an HCMP. The reason is that the relative differences between the evaluated schedulers remains unaffected when changing the details of the DRAM model.

6 Evaluation

We evaluate the following three schedulers:

- The *random scheduler*, for each time slice, randomly selects the applications to run on the big core(s).
- The *reliability-optimized scheduler* optimizes SSER using the algorithm described in Section 4.

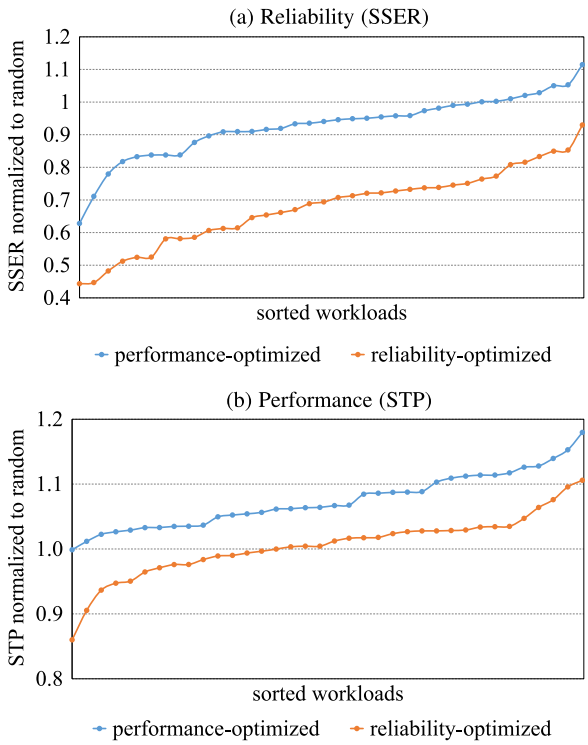

Fig. 6. System soft error rate (a) and system throughput (b) for reliability- and performance-optimized scheduling normalized to random scheduling for all four-program workloads on an HCMP with 2 big cores and 2 small cores.

- The *performance-optimized scheduler* optimizes system throughput [19] or weighted speedup, using the same sampling-based scheduling algorithm optimizing for STP rather than SSER.

In this section, we focus on the results for the 2B2S configuration. We refer the interested reader to the conference paper [1] for more detailed analysis and results across different core counts, asymmetric HCMP configurations and different clock frequencies for the big versus small cores. In addition, we also studied the impact of using only ROB ACE bits to steer scheduling as well as the impact of the sampling period.

6.1 2B2S Results

Fig. 6 evaluates system soft error rate and system throughput for the reliability- and performance-optimized schedulers, normalized to the random scheduler, for four-program workloads running on a 2B2S HCMP. SSER is a lower-is-better metric, while STP is a higher-is-better metric. Each dot represents a workload; the workloads are sorted by SSER and STP, respectively.

The reliability-optimized scheduler significantly and consistently improves reliability, i.e., SSER reduces by 32 percent on average and up to 55.6 percent compared to the random scheduler; and by 25.4 percent on average and by up to 60.2 percent compared to the performance-optimized scheduler. Reliability-aware scheduling effectively determines which applications are most vulnerable to soft errors and puts those applications on the small cores to improve overall system reliability.

The performance-optimized scheduler also reduces SSER over the random scheduler (by 7.3 percent on average). This

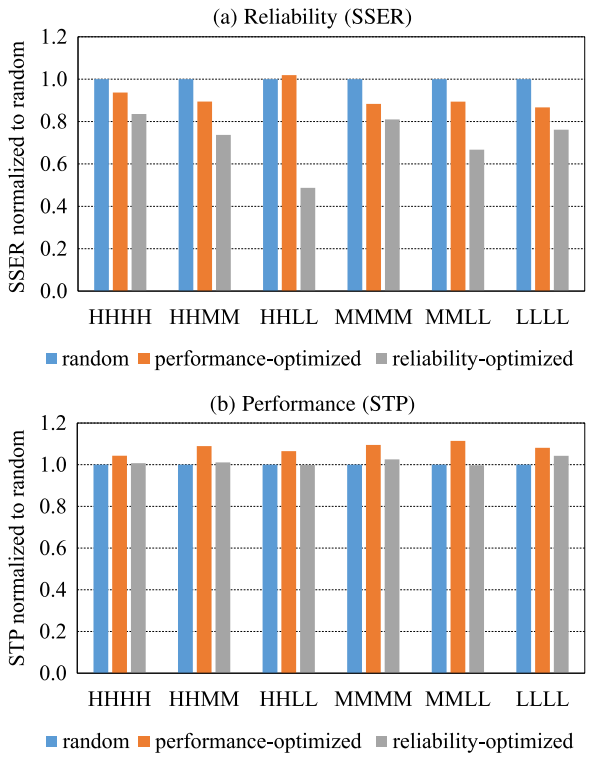

Fig. 7. SSER (a) and STP (b) on a 2B2S system per workload category.

improvement is substantially smaller and, moreover, it is not consistent, i.e., reliability decreases for a number of workloads. The reason for the (average) improved reliability is the apparent correlation between performance and reliability.

In terms of performance, the reliability-optimized scheduler yields similar performance to the random scheduler (half of the workloads are worse, half are better, resulting in an average near 0 percent difference), and degrades performance by only 6.3 percent on average (and by 18.7 percent at most) compared to the performance-optimized scheduler. The performance improvement of performance-optimized scheduling over random scheduling is in line with prior work [15].

6.2 Analysis by Workload Category

Fig. 7 shows the same results as Fig. 6 but now groups the results per workload category, with the categories defined based on big-core AVF, see Section 2.3. The largest improvement in system reliability is observed for the workload category that includes high-AVF applications and low-AVF applications (see HHLL). This does not come as a surprise: the high-AVF applications are scheduled on the small cores to reduce overall system reliability, while scheduling the low-AVF applications on the big cores. The workload categories with less divergent application behavior (HHMM and MMLL) also show substantial improvements in reliability, though not as high as for the HHLL category. Here, again, reliability-aware scheduling is able to schedule the applications with high AVF (relative to the other applications in the mix) on the small cores and vice versa. For the workload categories with similarly AVF-sensitive applications (all H, M or L applications), we observe modest improvement in reliability. The reliability-aware scheduler makes the correct scheduling decisions in terms of AVF, i.e., it schedules applications with the highest AVF on the small cores and

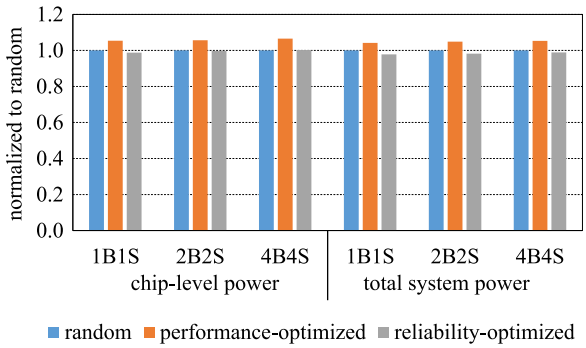

Fig. 8. Impact on chip-level and total system power consumption.

vice versa. Nevertheless, this leads to a small improvement in system reliability because of the lower system performance compared to performance-optimized scheduling, which tempers the improvement in soft error rate—remember that SSER weights relative per-application slowdown.

7 Performance versus Power Consumption versus Reliability

There is an important trade-off between performance, power and reliability, as corroborated by a recent study [24]. In the previous section, we focused on performance and reliability. However, changing the workload schedule on a heterogeneous multicore also affects power consumption. Therefore, in this section, we first evaluate how reliability-aware scheduling affects power consumption. We then subsequently explore the trade-off between scheduling for performance, power and reliability.

7.1 Impact of Reliability-Aware Scheduling on Power

Fig. 8 quantifies the impact on chip-level power (including L3) and total system power (processor plus DRAM) with increasing core count. We use McPAT [25] to quantify power consumption. The bottom line is that reliability-optimized scheduling reduces chip-level and system power by 6 and 6.2 percent on average, respectively, relative to performance-optimized scheduling. The reason is that performance-optimized scheduling puts applications on a big core for performance reasons although this may increase power consumption. For example, a memory-intensive application with high degrees of MLP will be scheduled on the big core to improve performance [15]; this will lead to an increase in power consumption. The reliability-aware scheduler on the other hand schedules this workload on the small core to reduce soft error vulnerability, also reducing power.

7.2 Trade-Offs in Performance-, Power- and Reliability-Optimized Scheduling

We implemented a *power-optimized scheduler* to evaluate the impact of optimizing for power on reliability. The power-optimized scheduler, alike our reliability- and performance-optimized schedulers, is a sampling-based scheduler. The scheduling decision is based on the energy consumed by each core per quantum. Note that an ideal power- or reliability-optimized schedule can be achieved by scheduling workloads on small cores in a sequential manner. However, all our scheduling policies maintain the fundamental

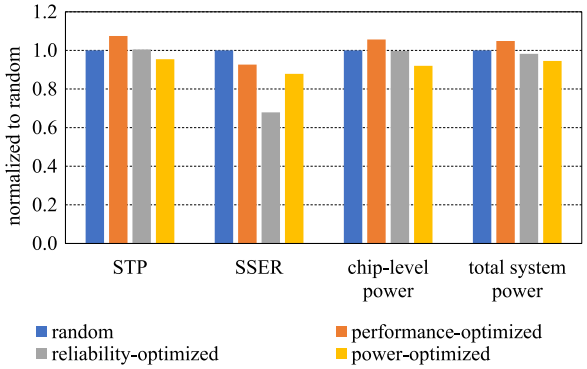

Fig. 9. Comparing performance-, reliability- and power-optimized schedulers for all four-program workloads on an HCMP with 2 big cores and 2 small cores. All results are normalized to the random scheduler.

assumption that no core remains idle while an HCMP is executing a multiprogram workload. Fig. 9 shows the relationship among power, performance and reliability when we execute four-program workloads on 2 big and 2 small cores. Optimizing power always leads to performance degradation, and also leads to an overall improvement in reliability (by 12.2 percent on average) compared to random scheduling. Benchmarks such as `cactusADM` and `hmmer` expose a large state inside the big core, causing high vulnerability and power consumption. Such benchmarks are scheduled on a small core to improve both power and reliability. For such benchmarks, optimizing for power also leads to an improvement in reliability, and vice versa, optimizing reliability also improves power.

There are several workloads for which a reliability-optimized schedule is different from a power-optimized schedule. For example, `milc` and `sjeng` run on different core types for reliability and power when they co-run. Compared to `milc`, `sjeng` incurs much higher power on the big core compared to the small core. Therefore, for power, it is always scheduled on the small core. On the other hand, `milc` is a memory-intensive benchmark that maintains a large vulnerable state inside the core while waiting for long-latency memory requests to complete. Since the difference in SER for `sjeng` is small between the big and small cores, the reliability-optimized scheduler runs `milc` on the small core and `sjeng` on the big core.

The key take-away from the results reported in Fig. 9 is that there is a trade-off between performance-, power- and reliability-optimized scheduling. Performance-optimized scheduling leads to high performance, but also leads to high power consumption and soft error vulnerability. Power-optimized scheduling minimizes power consumption, however this comes at a cost in performance. Reliability to soft errors slightly improves under power-optimized scheduling compared to performance-optimized scheduling. Reliability-optimized scheduling improves reliability by a significant margin while being on par with random scheduling in terms of performance and power consumption.

8 Reliability-Aware Scheduling Under Performance Constraints

So far, we assumed that the goal is to optimize reliability while considering performance after the fact, i.e., we schedule applications to core types to optimize for reliability and we pay the cost this may incur in terms of performance. In many systems however, performance is more important than reliability, and one may not be willing to pay an average 6.3 percent performance degradation compared to performance-optimized scheduling, even if this improves reliability by 25.4 percent on average, as previously reported. Although reliability is an important concern, one may not want to incur a performance hit by more than a predefined limit, say 2 percent, but within this constraint one may yet want to improve reliability. In this section, we explore reliability-aware scheduling under performance constraints.

With a minimum acceptable performance level specified, we propose to augment the scheduler with a mechanism to dynamically switch between the reliability- and performance-optimized modes at runtime. The decision to choose either of the two modes depends on the requirements of the system and the workload under execution. If reliability is of utmost importance—for example, in systems working at higher altitudes in space—the goal should be to optimize for reliability. In such cases, running in the reliability-optimized mode suits the best. However, when performance is the key concern and performance is not allowed to drop below a certain performance level relative to performance-optimized scheduling, the scheduler should switch to the performance-optimized mode once the performance is about to drop below the specified level.

8.1 Scheduling Mechanism

To achieve performance above a specified level, we need to keep track of performance while improving reliability. At the end of every scheduling quantum, we estimate performance (i.e., STP) for all possible schedules and discard the schedules not meeting our performance criterion. Of the remaining schedules, we choose the schedule with the lowest SSER as the schedule for the next quantum. If an application continues to run on a particular core type for 10 consecutive scheduling quanta (1 ms each), a sampling phase (0.1 ms) is triggered to account for the possibility of a phase change in the application behavior.

8.2 Evaluation

To evaluate reliability-aware scheduling under performance constraints, we consider reliability and performance for four-program workloads on a 2B2S system under various performance constraints, see Fig. 10. We start with the performance-optimized scheduler (shown on the left) and gradually increase the allowable performance degradation. Eventually, when there is no performance constraint, we end up with the reliability-optimized scheduler (shown on the right). When the performance limit is set at x%, the STP of the four-program workload must never be degraded by more than x% at any point during the execution compared to the performance-optimized schedule. For example, when the performance limit is set at 4 percent, and the highest possible STP of a four-program workload in a scheduler quantum is 3.0 on a 2B2S system, then the scheduler should map applications in such a manner that the STP does not degrade below 2.88 (4 percent degradation of 3.0). This constraint is met every quantum and ensures that the workload will not experience an overall performance degradation by

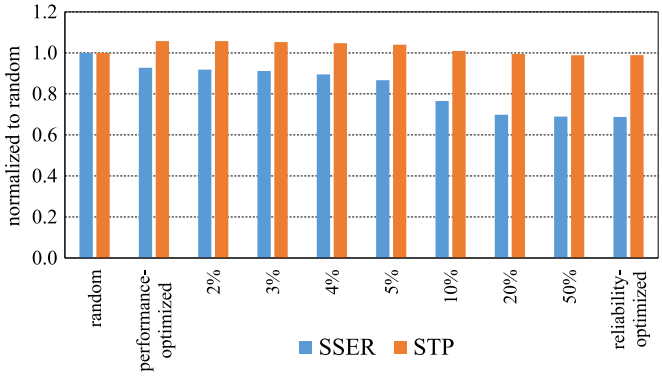

Fig. 10. Average reliability (SSER) and performance (STP) relative to the random scheduler for reliability-aware scheduling under performance constraints, for four-program workloads on a 2B2S system.

more than 4 percent; in fact the average performance degradation is typically smaller than the performance limit that was set.

The results in Fig. 10 indicate a clear trend—increasing the allowable performance limit increases the performance degradation while at the same time improving reliability, as expected. At small performance limits, the improvement in reliability is limited and so is the impact on performance. The reason for the small impact is the limited number of opportunities for choosing an alternative schedule. In particular, there are six possible mappings for a four-program workload on two big and two small cores: `BBSS`, `SSBB`, `BSBS`, `SBSB`, `BSSB` and `SBBS`; where a `B` and `S` represents the respective application running on the big versus small core, respectively. One of these schedules is the performance-optimized schedule. The scheduler has limited opportunity to choose a schedule other than the one that optimizes performance while remaining within 2 percent of the performance-optimal schedule. However, it may still successfully pick such a schedule in very few cases.

Increasing the performance limit provides more flexibility to the scheduler and the improvement in reliability is also higher. In particular, the average gain in SSER for a limit of 5 and 10 percent equals 13.5 and 23.5 percent, respectively. Note that performance is still better than the random scheduler for these performance levels. As the limit is further increased, the scheduler starts to choose schedules that are more similar to the ones chosen by the reliability-optimized scheduler. For the 20 and 50 percent performance limits, the numbers are very close to the reliability-optimized scheduler—an average improvement in reliability of 32 percent at the cost of a 1 percent performance degradation compared to the random scheduler. Overall, we conclude that the improvement in reliability is always much higher than the degradation in performance. The higher the allowable performance degradation, the higher is the improvement in reliability; the actual limit, however, can be adjusted by the system administrator or end user based on the requirements.

9 Multi-Threaded Workloads

So far, we considered multiprogram workloads composed out of single-threaded programs, for which we observed the highest improvements in reliability for workload mixes consisting of diverse applications, i.e., high-AVF applications running concurrently with low-AVF applications; the smallest improvements are observed for workload mixes composed out of applications with similar AVF characteristics. We now consider multi-threaded workloads. Most multi-threaded workloads are data-parallel in which all threads execute the same code on different portions of the data. As a result, all threads exhibit similar execution behavior. We refer to these workloads as *homogeneous* workloads. Some multi-threaded workloads however expose pipelined parallelism, i.e., the outcome produced by one thread is the input for another thread. These workloads are *heterogeneous*, i.e., different threads execute different code. Based on the results obtained for the multi-program workloads, we expect limited improvement for the multithreaded workloads that are homogeneous, but we expect a higher improvement for the heterogeneous workloads.

9.1 Metrics

For multiprogram workloads, the necessity for metrics such as STP for performance and SSER for reliability arises from the fact that co-executing programs affect each other's performance. However, for multithreaded workloads, execution time or *start-to-finish* time correctly measures performance, i.e., this is the time it take to get a unit of work done. Similarly, since the amount of work performed by a multi-threaded program is fixed, SER is an appropriate metric to quantify the vulnerability of a multi-threaded program to soft errors. ABC of a multithreded program is the sum of the ABC values for all threads. Once we know the overall ABC, SER can be calculated as described in Section 2.

9.2 Performance-Optimized Scheduling

Identifying bottlenecks and improving performance of multithreaded workloads on multicore hardware is a challenging task, especially on heterogeneous multicore processors, and a number of prior works have focused on this problem, see for example [26], [27], [28]. The challenge when executing multi-threaded workloads on multicore hardware is to make sure that all threads make equal progress, i.e., all threads need to reach the end of the execution or the next barrier at roughly the same time, or in other words, the execution needs to be balanced. This may be complicated because of negative interference in shared resources, e.g., one thread may kick out another thread's data from the shared cache. Heterogeneous multicore processors further complicate this, i.e., the thread(s) running on the big core(s) make much faster progress than the one(s) running on the small core(s). One solution is to make sure all threads get an equal share of the big core cycles, i.e., by allowing all threads to run on a big core alternately. This leads to a balanced execution, improving overall application performance. This is typically a viable solution for homogeneous multi-threaded workloads, however, heterogeneous workloads need a more involved solution, i.e., we need to make sure all threads make equal progress, as described by Van Craeynest et al. [29]. There is a subtle but important difference between *equal share* and *equal progress*. Equal share guarantees the same number of big core cycles for all threads; equal progress on the other hand guarantees that all threads benefit equally from running on the big cores, e.g., if one thread benefits twice as much from running on

TABLE 3
Multithreaded Benchmarks from PARSEC and Rodinia

Suite	Benchmark	Input size
Rodinia	backprop	large
	bfs	large
	cfd	large
	hotspot	large
	kmeans	large
PARSEC	bodytrack	large
	canneal	large
	dedup	medium
	ferret	medium
	fluidanimate	large
	swaptions	large

the big core, it will receive only half as many cycles. Equal progress enables balanced execution even for heterogeneous multi-threaded workloads. Van Craeynest et al. [29] find that equal-progress scheduling is the best performing performance-optimized scheduler, which we adopt accordingly in this section.

9.3 Results

For our evaluation, we compare the reliability- and performance-optimized schedulers on an HCMP with two big and two small cores (2B2S). The two schedulers minimize SER and total execution time, respectively. We also compare against a random scheduler that randomly selects threads to run on the big cores.

9.3.1 Methodology

We need to consider a few subtle changes in the experimental methodology for the multi-threaded workloads in comparison to the multiprogram workloads. The scheduling quantum can be fixed for the multiprogram workloads (e.g., 1 ms). However, this is not appropriate for multi-threaded programs for which the number of running threads may vary dynamically at runtime because of sequential code sections and synchronization activity. Therefore, a scheduler should only take into account the threads performing useful work. When the number of active threads does not change during the course of a 1 ms time interval, we fix the scheduling quantum to 1 ms. In addition, a quantum starts (or ends) when a thread changes from running to waiting and vice versa. In such cases, the size of a quantum will be less than 1 ms. This flexibility in quantum size is required to consider all running threads for scheduling. Sampling is performed in a manner similar to what is done for the multiprogram workloads—when a thread continues to run for ten consecutive quanta on one core type, we trigger the sampling phase for a period of 0.1 ms.

Another difference is that the number of active threads at runtime can be less than the number of cores available in an HCMP. This may lead to certain cores remaining idle for some time during program execution. When there is a possibility of a core remaining idle during a scheduler quantum, we utilize as many big cores as possible to take advantage of their high performance. For example, in a 2B2S system, if there are only three active threads during a quantum, the two big cores will always be running two threads and one small core will run the third thread leading to one small core remaining idle.

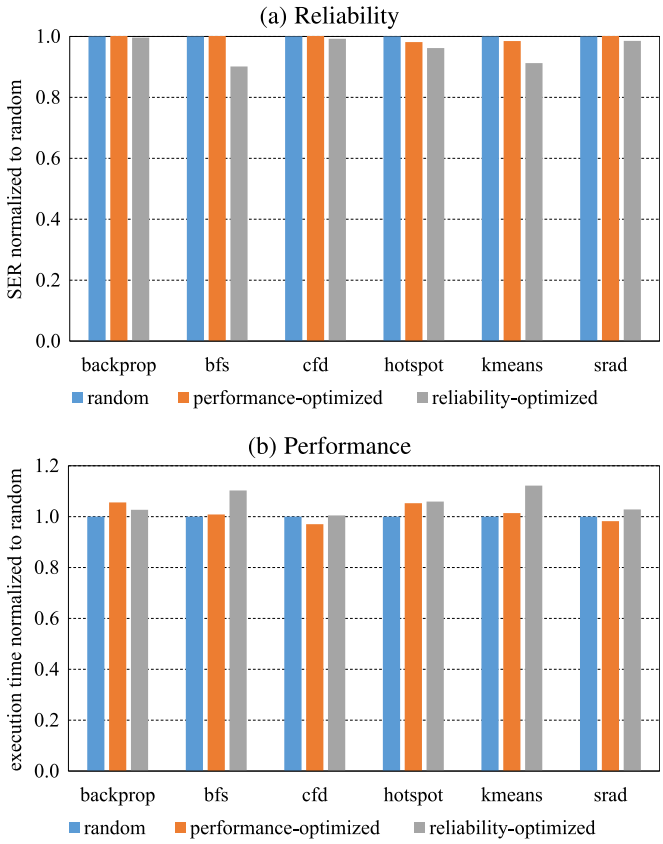

Fig. 11. Reliability (a) and performance (b) for the Rodinia benchmarks on a 2B2S system.

We use benchmarks from the Rodinia [30] and PARSEC [31] suites to evaluate our reliability-aware scheduler for multithreaded workloads, see Table 3. We simulate the benchmarks that we were able to successfully run on our simulator. We consider the parallel portion of the benchmarks in the evaluation; the sequential phases are run on the big core for highest performance. All benchmarks except for `ferret` were executed on a 2B2S system. `ferret` requires at least six threads (cores) for execution and therefore we simulated six threads for `ferret` on an HCMP with three big and three small cores (3B3S).

9.3.2 Rodinia

Fig. 11 shows reliability and performance for the Rodinia benchmarks for reliability-aware scheduling compared to random and performance-optimized scheduling. The highest improvement in soft error rate compared to both random and performance-optimized scheduling is achieved for `bfs` (10 percent), followed by `kmeans` (8.8 percent). The improvement is less significant for the other benchmarks. Looking at performance, we observe that reliability-aware scheduling is either performance neutral or degrades performance. We note a one-to-one trade-off between reliability and performance for most benchmarks. For example, for `bfs`, reliability-aware scheduling improves reliability by 10 percent while at the same time degrading performance by 10 percent. The reason is that the Rodinia benchmarks are homogeneous data-parallel workloads, and hence there is limited opportunity to

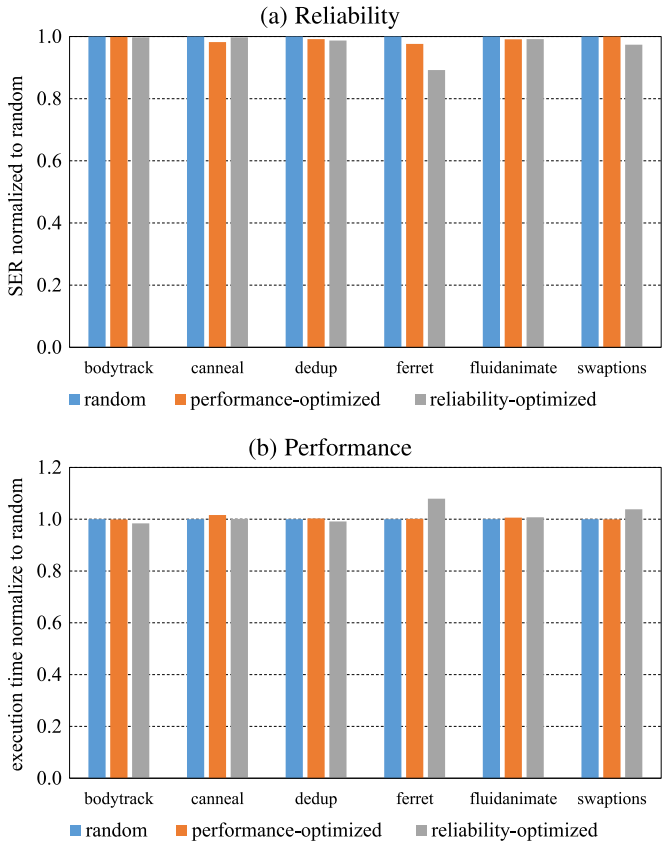

Fig. 12. Reliability (a) and performance (b) for the PARSEC benchmarks.

improve reliability, as expected and argued above. The data-parallel nature of the workloads is also the reason for similar performance between the random and performance-optimized scheduling. For some workloads (for example, `cfd` and `srad`), the performance-optimized scheduler performs slightly worse than the random scheduler because of sampling overhead. Our sampling-based scheduler ensures that every thread gets to run on both core types in the beginning, followed by periodic sampling every 10 quanta. This leads to slight disturbances and a small degradation in performance compared to the random scheduler.

9.3.3 PARSEC

Fig. 12 shows similar results for the PARSEC benchmarks. Two of the PARSEC benchmarks are heterogeneous workloads, namely `ferret` and `dedup`; all other benchmarks are homogeneous data-parallel workloads. We observe similar results for the homogeneous PARSEC benchmarks as for the Rodinia benchmarks: the improvement in reliability through reliability-aware scheduling leads to an almost equally high degradation in performance (and both are small). For one of the heterogeneous workloads, namely `ferret`, we do observe an interesting result: the improvement in reliability (11 percent) is higher than the degradation in performance (7 percent), which is in line with the results and conclusion obtained for the multiprogram workloads. Unfortunately, we do not observe a similar result for `dedup`, the other heterogeneous benchmark. Through detailed analysis using *bottle graphs* [32] of `dedup`'s execution behavior (not shown because of space constraints), we observe that there is a very high degree of parallel imbalance among the threads. One critical thread runs for a longer time than all other threads put together. This leads to the other non-critical threads remaining idle for most of the execution. Since our scheduling policy never leaves a big core idle, the critical thread is always running on the big core for all three schedulers, thus leading to similar reliability and performance figures for all of them.

The overarching conclusion from this section is that reliability-aware scheduling has limited benefit for multi-threaded workloads. The primary reason is that different threads typically execute the same code and hence there is limited opportunity to exploit diversity in AVF characteristics across the different threads. We typically observe a one-to-one trade-off between reliability and performance. Only in a limited number of cases, i.e., heterogeneous workloads with different threads that execute different code and that exhibit different AVF characteristics, do we observe an opportunity to improve reliability at the expense of a relatively small performance degradation.

10 Incorporating Unprotected L1 Caches

Protection techniques based on Error Detecting Codes (EDC) and Error Correcting Codes (ECC) incur chip area, power and possibly latency overheads, and are typically applied to the cache levels beyond the L1 caches [33]. Several prior works estimate and mitigate soft errors of on-chip caches in general, and L1 caches in particular, see for example [5], [33], [34], [35], [36], [37]. Recent work also focuses on dynamically reconfiguring last-level caches and improving reliability across cache hierarchy in the presence of multibit soft errors [38], [39], [40]. Reliability-aware scheduling as proposed in this paper works for the case in which the L1 caches are protected (which is what we assumed so far) as well as for the case in which the L1 caches are not protected (which is the subject of this section). In order for reliability-aware scheduling to be able to incorporate L1 cache soft error vulnerability, we need to also estimate and measure L1 cache soft error vulnerability. In this section, we first explain our methodology to dynamically compute the ACE Bit Count for the L1 caches and then evaluate how well reliability-aware scheduling performs taking into account reliability of the core *and* L1 caches.

10.1 Estimating Cache Soft Error Vulnerability

Our methodology for calculating ACE Bit Count in the data and tag arrays is based on the work done by Biswas et al. [5]. A cacheline is ACE if its correctness is required for the correct execution of a program. (Note we assume both the L1 D-cache and L1 I-cache to be write-back caches.) For the data array, ABC can be estimated as follows. There are four time intervals during which a cacheline is ACE: *fill-to-read*, *read-to-read*, *write-to-read* and *write-to-evict*. For the tag array, the correctness of a program is affected only by the *false-positive* case, when an incorrect cacheline is returned due to an error in the tag bits. Therefore, to estimate ABC in this case, we implement the *hamming-distance-one* analysis and conservatively assume that all (tag) entries of a set are at a hamming-distance of 1 from the tag bits of the requested memory address.

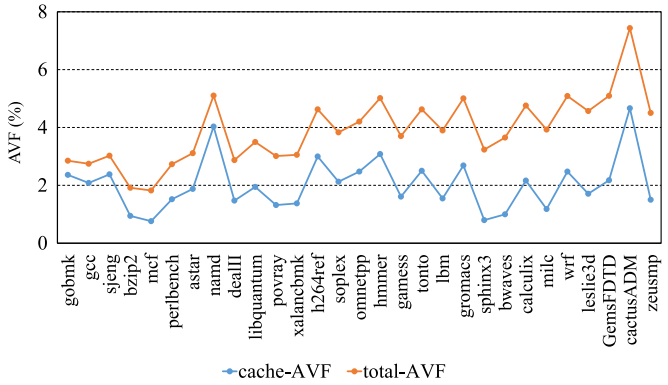

Fig. 13. Cache-AVF and total-AVF for the SPEC CPU2006 benchmarks on a big out-of-order core.

10.2 Hardware Overhead

The hardware cost for computing ABC for the L1 caches is limited. There are 512 64 B cachelines in a 32 KB L1 cache. Assuming a quantum size of 1 ms, this amounts to 375,940 cycles when running at 2.66 GHz. This is also the maximum number of cycles a cacheline can be ACE. Accounting for this many cycles requires 18.5 bits; assume 20 bits per cacheline. For each cacheline, we keep track of the last access time. This amounts to 20 bits per cacheline ABC counter, or a total of 1,280 bytes.

Whenever a cacheline is read/written/evicted, we update one global cache-wide ABC counter. In the 'worst' case, the entire cache can be ACE for one quantum, which implies that this cache-wide ABC counter requires 36 bits. When a cacheline is read or evicted, we add the difference between the current cycle count (since the beginning of the scheduling quantum) and the cacheline ABC counter to the global counter, and we replace the cache ABC counter value with the current cycle count upon a read, eviction and write. A 36-bit adder is equivalent to 250 bits, similar to what is described in Section 4.2. The addition of ABC counters across quanta can be done in software. The overall hardware cost for an L1 cache amounts to 1,314 bytes.

10.3 Impact of Caches on Soft Error Vulnerability

The impact L1 caches have on soft error vulnerability is quantified in Fig. 13: cache-AVF and total-AVF (that is, AVF for core plus L1 caches; total-ABC is defined similarly) are shown for the SPEC CPU2006 benchmarks; the benchmarks are sorted in the same order as in Fig. 1. (Note that the reported AVF values are much smaller in Fig. 13 compared to Fig. 1; this is because the L1 caches are now included in the total structure size.) It is clear from the figure that there is a strong correlation between total-AVF and cache-AVF. This is primarily because total-ABC is dominated by the L1 caches. The size of, or more precisely the architecture state contained in, the L1 caches is ten times higher than the out-of-order core—64 KB versus almost 6 KB. In spite of the strong correlation between cache-AVF and total-AVF, we observe that the gap between both curves widens going from left to right in Fig. 13. This is because the benchmarks on the right-hand side of the graph have higher core-AVF, as previously reported.

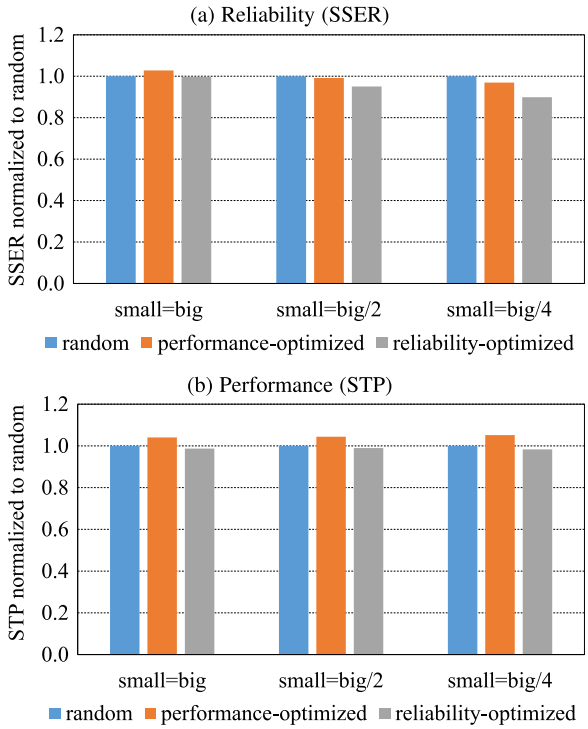

Fig. 14. SSER (a) and STP (b) of four-program workloads on a 2B2S system with decreasing L1 cache size for the small cores.

10.4 Results

Fig. 14 shows results for reliability-aware scheduling when ABC for the L1 caches is also taken into account. That is, total-ABC is used in Algorithm 1 as well as in Equation 6 for estimating SSER. We evaluate three cases while varying the size of the L1 caches for the small core. In the first case, the L1 caches for both the big and small cores are equal in size. In other two cases, we reduce the size of the L1 caches for the small core by a factor of 2 and 4, respectively. When the L1 cache size is equal between the big and small core, the impact on reliability and performance is small. The reason is twofold: (i) the total amount of vulnerable state is dominated by the L1 caches, as described above, and (ii) execution time on the small core takes longer and as a result cachelines get exposed to soft errors for a longer duration, further narrowing the difference in vulnerable state between the big and small cores. Reducing the size of the L1 caches in the small core, the difference in vulnerable state increases between the big and small cores, which leads to significant average improvements in SSER by 5 and 11 percent for half the cache size and a quarter the cache size for the small cores, respectively. Note that performance is largely unaffected compared to random scheduling. These results demonstrate that reliability-aware scheduling is beneficial even if the L1 caches are unprotected and need to be taken into account as part of the scheduling policy. In addition, reliability-aware scheduling is more effective as cache size differs between the big and small cores.

11 Related Work

We now discuss related work in processor reliability, as well as recent work in scheduling for HCMPs.

11.1 Monitoring, Modeling and Improving Reliability

Processor reliability is a growing concern, and a significant body of prior work targets decreasing the occurrence of soft errors, either through radiation-hardened circuit design [6], error detection and correction mechanisms [7], or architectural techniques [8], [9]. Our scheduling technique is orthogonal to these approaches, and provides additional reliability improvements.

Other researchers have studied monitoring and modeling reliability for processor design (e.g., where to add error detection) and online reliability estimation (e.g., to find out when to enable an architectural error reduction technique that may also incur a performance hit). One way to evaluate soft error reliability is through fault injection, and to monitor what fraction of faults lead to incorrect program executions [41]. Mukherjee et al. [3] propose ACE bit analysis as an alternative to fault injection to evaluate the reliability in architecture studies. They also introduce the concept of AVF. Biswas et al. [5] show how to measure AVF for address-based structures. Sridharan and Kaeli [42] propose to split AVF into PVF (program vulnerability factor) and HVF (hardware vulnerability factor), which can be determined independently. Other prior work models AVF through regression on performance counters [43], [44], or through analytical mechanistic modeling [22]. Nair et al. [45] develop a methodology for creating AVF-stressing benchmarks, providing a processor AVF upper bound.

No prior work has studied reliability characteristics of HCMPs, or has considered HCMP scheduling as a way to improve reliability. We are also the first to propose a system-level reliability metric for multiprogram workloads.

11.2 Scheduling Heterogeneous Multicores

Kumar et al. [10], [11] advocate single-ISA heterogeneous multicores to improve energy and power efficiency. Many proposals advocate scheduling compute-intensive applications on the big cores, because they show the highest performance improvement [46], [47], [48]. Van Craeynest et al. [15] show that memory-intensive applications can also show important performance gains on big cores if they are able to exploit more memory-level parallelism. Other proposals focus on optimizing energy efficiency [16] or power efficiency [17], [18]. Recent proposals [49] exploit redundant multithreading to improve overall dependability of multicore systems by mapping tasks onto different cores. We are the first to improve reliability on HCMPs through scheduling.

12 Conclusion

Applications exhibit different soft error reliability characteristics on big, out-of-order cores versus small, in-order cores. This provides considerable opportunity to improve system reliability through scheduling on HCMPs. We propose a reliability-aware scheduler that samples the reliability characteristics of running applications on either core type, and dynamically schedules applications on big versus small cores to improve overall system reliability. We propose a novel system-level reliability metric, system soft error rate, that weights per-application SER by their relative slowdown to account for the difference between small and big core performance. The proposed scheduler leverages a low-overhead (296 bytes per core) counter architecture to track hardware occupancy.

Reliability-aware scheduling improves system reliability by 25.4 percent on average and up to 60.2 percent compared to performance-optimized scheduling, while degrading performance by 6.3 percent only. Moreover, as a side effect, reliability-aware scheduling reduces power consumption by 6.2 percent on average compared to performance-optimized scheduling. We evaluate the trade-off between reliability-, power- and performance-optimized scheduling; we demonstrate reliability-aware scheduling under performance constraints; we evaluate reliability-aware scheduling for multi-threaded workloads; and we demonstrate the ability to take unprotected L1 caches into account.

Acknowledgments

This paper is an extension of "Reliability-Aware Scheduling on Heterogeneous Multicore Processors" by the same authors [1], presented at the 2017 International Symposium on High-Performance Computer Architecture (HPCA). This paper includes several novel contributions over the HPCA paper including (i) trade-off analysis between reliability-, power- and performance-optimized scheduling, (ii) reliability-aware scheduling under performance constraints, (iii) reliability-aware scheduling for multi-threaded workloads, and (iv) reliability-aware scheduling while taking into account soft error vulnerability in the L1 caches.

References

[1] A. Naithani, S. Eyerman, and L. Eeckhout, "Reliability-aware scheduling on heterogeneous multicore processors," in *Proc. 23rd IEEE Symp. High Perform. Comput. Archit.*, 2017, pp. 397–408.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[3] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 29–40.

[4] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 389–398.

[5] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," in *Proc. 32nd Annu. Int. Symp. Comput. Archit.*, 2005, pp. 532–543.

[6] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron CMOS technology," *IEEE Trans. Nucl. Sci.*, vol. 43, no. 6, pp. 2874–2878, Dec. 1996.

[7] M. Nicolaidis, "Design for soft error mitigation," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 405–418, Sep. 2005.

[8] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam, "Mechanisms for bounding vulnerabilities of processor structures," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 506–515.

[9] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 264–275.

[10] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. 36th Int. Symp. Microarchit*, 2003, pp. 81–92.

[11] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 64–75.
[12] P. Greenhalgh, "Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms," 2011. [Online]. Available: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf
[13] NVidia, "Variable SMP – a multi-core CPU architecture for low power and high performance," 2011. [Online]. Available: http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-%Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf
[14] N. Chitlur, et al., "QuickIA: Exploring heterogeneous architectures on real prototypes," in *Proc. Int. Symp. High Perform. Comput. Archit*, 2012, pp. 1–8.
[15] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 213–224.
[16] A. Lukefahr, et al., "Composite cores: Pushing heterogeneity into a core," in *Proc. ACM/IEEE Int. Symp. Microarchit.*, 2012, pp. 317–328.
[17] T. S. Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 161–176.
[18] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications," in *Proc. 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 137–149.
[19] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May./Jun. 2008.
[20] R. Uma, V. Vijayan, M. Mohanapriya, and S. Paul, "Area, delay and power comparison of adder topologies," *Int. J. VLSI Des. Commun. Syst.*, vol. 3, no. 1, pp. 153–168, 2012.
[21] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, 2014, Art. no. 28.
[22] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John, "A first-order mechanistic model for architectural vulnerability factor," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 273–284.
[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2002, pp. 45–57.
[24] K. Swaminathan, N. Chandramoorthy, C. Cher, R. Bertran, A. Buyuktosunoglu, and P. Bose, "Bravo: Balanced reliability-aware voltage optimization," in *Proc. 23rd IEEE Symp. High Perform. Comput. Archit.*, 2017, pp. 97–108.
[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 469–480.
[26] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proc. Int. Symp. Comput. Archit.*, 2009, pp. 290–301.
[27] J. Joao, M. Suleman, O. Mutlu, and Y. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 223–234.
[28] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proc. Int. Symp. Comput. Archit.*, 2013, pp. 511–522.
[29] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 177–188.
[30] S. Che, et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[31] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.
[32] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2013, pp. 355–372.
[33] D. Sorin, *Fault Tolerant Computer Architecture*. San Rafael, CA, USA: Morgan and Claypool Publishers, 2009.
[34] A. Biswas, et al., "Explaining cache SER anomaly using DUE AVF measurement," in *Proc. 16th IEEE Symp. High Perform. Comput. Archit.*, 2010, pp. 1–12.
[35] S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: Myth or necessity?" in *Proc. 10th IEEE Pacific Rim Int. Symp. Dependable Comput.*, 2004, pp. 37–42.
[36] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli, "Balancing performance and reliability in the memory hierarchy," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2005, pp. 269–279.
[37] S. Wang, J. Hu, and S. G. Ziavras, "On the characterization and optimization of on-chip cache reliability against soft errors," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1171–1184, Sep. 2009.
[38] F. Kriebel, S. Rehman, A. Subramaniyan, S. J. B. Ahandagbe, M. Shafique, and J. Henkel, "Reliability-aware adaptations for shared last-level caches in multi-cores," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 4, pp. 1–26, Aug. 2016.
[39] F. Kriebel, A. Subramaniyan, S. Rehman, S. J. B. Ahandagbe, M. Shafique, and J. Henkel, "R2cache: Reliability-aware reconfigurable last-level cache architecture for multi-cores," in *Proc. Int. Conf. Hardw. Softw. Codesign Syst. Synthesis*, 2015, pp. 1–10.
[40] A. Subramaniyan, S. Rehman, M. Shafique, A. Kumar, and J. Henkel, "Soft error-aware architectural exploration for designing reliability adaptive cache hierarchies in multi-cores," in *Proc. Int. Conf. Des., Autom. Test Eur.*, 2017, pp. 37–42.
[41] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 341–352.
[42] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance AVF analysis," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 461–472.
[43] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 516–527.
[44] D. Lide, L. Bin, and P. Lu, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics," in *Proc. 15th Int. Symp. High Perform. Comput. Archit.*, 2009, pp. 129–140.
[45] A. A. Nair, L. K. John, and L. Eeckhout, "AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," in *Proc. 43rd Annu. Int. Symp. Microarchit.*, 2010, pp. 125–136.
[46] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proc. 46th Annu. Des. Autom. Conf.*, 2009, pp. 927–930.
[47] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 125–138.
[48] D. Shelepov, et al.,"HASS: A scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.
[49] K. Chen, J. Chen, F. Kriebel, S. Rehman, M. Shafique, and J. Henkel, "Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3441–3455, Nov. 2016.

Ajeya Naithani received the MS degree in computer science from the University of Arizona, in 2011. He is working toward the PhD degree with Ghent University, Belgium. His research interests include the area of computer architecture with an emphasis on designing techniques to improve soft error reliability of processors.

Stijn Eyerman received the MSc and PhD degree from Ghent University, in 2004 and 2008, respectively. He is currently working as a research scientist for Intel. He has published more than 40 papers at conferences and journals, two of which have been awarded with an IEEE Micro Top Picks selection. His interests include processor performance modeling and scheduling on (heterogeneous) multicore processors.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University, in 2002. He is professor with Ghent University, Belgium. His research interests include the area of computer architecture, with a specific interest in performance analysis, evaluation and modeling. He is the current editor-in-chief of the *IEEE Micro (2015-2018)*, and is the recipient of the 2017 ACM SIGARCH Maurice Wilkes Award. His research is funded by the European Research Council under the European Communitys Horizon 2020 Programme/ERC Advanced Grant agreement no. 741097, as well as Research Foundation – Flanders (FWO) grants no. G.0434.16N and G.0144.17N.