# Reliability-Aware Scheduling on Heterogeneous Multicore Processors

Ajeya Naithani
Ghent University, Belgium

Stijn Eyerman*
Intel, Belgium

Lieven Eeckhout
Ghent University, Belgium

## 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% on average (and up to 60.2%) compared to performance-optimized scheduling on a heterogeneous multicore processor with two big cores and two small cores, while degrading performance by 6.3% only. We also introduce a novel system-level reliability metric for multiprogram workloads on (heterogeneous) multicores. We further show that our reliability-aware scheduler is robust across core count, number of big and small cores, and their frequency settings. The hardware cost in support of our reliability-aware scheduler is limited to 296 bytes per core.

## 1. INTRODUCTION

As technology shrinks and operation voltages decrease, the amount of charge in the transistors' gates 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 [1, 16, 24]. 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 [2, 3, 16, 20, 25, 30].

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 [8], NVidia's Tegra [21], and Intel's QuickIA [6]. Prior work in scheduling for HCMPs focused on optimizing performance [28], energy efficiency [15], and power efficiency [17, 31]. However, no prior work has explored scheduling for reliability on HCMPs.

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% on average (and up to 55.6%) 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% on average (and up to 60.2%), while degrading performance by 6.3% only. We show that our scheduler performs well across core count, number of big versus small cores, and frequency settings.

Overall, we make the following key contributions in this paper:

- We analyze the difference in reliability characteristics between big and small cores.
- We show the potential for optimizing reliability through scheduling on HCMPs.

*This work was done while at Ghent University.

- We define a novel metric, System Soft Error Rate (SSER), for assessing reliability to soft errors for multiprogram workloads on (heterogeneous) multicores.
- We propose a dynamic online scheduler to optimize reliability in HCMPs.

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. Finally, we describe related work (Section 7) and conclude (Section 8).

## 2. MOTIVATION

In this section, 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.

The *architectural vulnerability factor (AVF)* [16] 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, ACE bit count is therefore proportional to SER.

### 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, leading to an opportunity for improving reliability through scheduling.

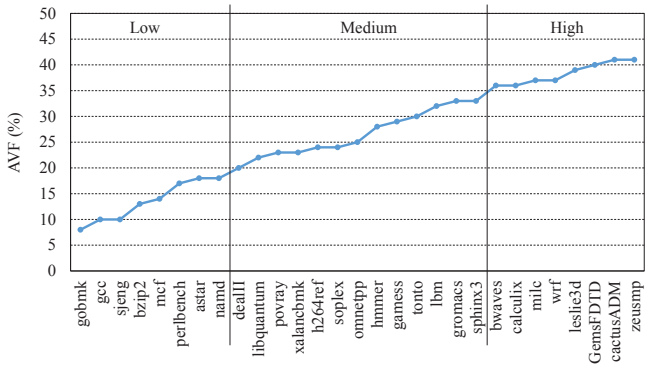


**Figure 1: AVF for the SPEC CPU2006 benchmarks (sorted) on a big out-of-order core.**

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, the functional units, the issue queue, the 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 of the processor.
- 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, wrong-path instructions). The smaller the occupied fraction is, the smaller is the error probability. This second contributor thus 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 structures than the small core. As a result, a big core is likely to expose more vulnerable state than an in-order 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. Figure 1 shows (sorted) AVF for the SPEC CPU2006 benchmarks on a big out-of-order core. (See Section 5

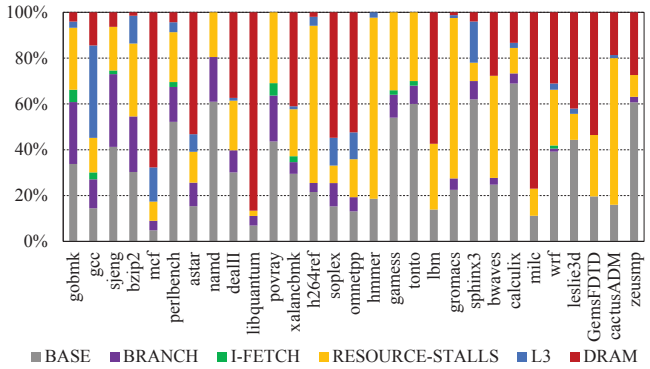


**Figure 2: Normalized CPI stacks for the SPEC CPU2006 benchmarks on a big out-of-order core.**

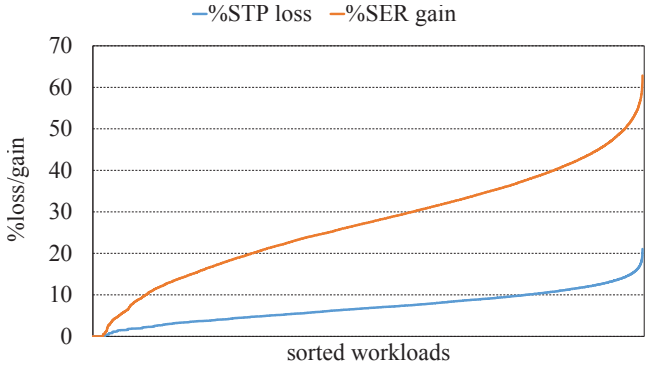


**Figure 3: Percentage STP loss and SER gain for an oracle reliability-optimized scheduler relative to a performance-optimized scheduler for four-program workloads on an HCMP with two big cores and two small cores.**

for details regarding our experimental setup.) 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 ROB for 16 cycles, this amounts to 1024 ACE bits. This way of measuring incorporates structure size, occupancy and execution time. 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, 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 versus small core is not as big, 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. Figure 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 in the evaluation section.

It is interesting to relate the AVF graph to the normalized CPI stacks shown in Figure 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 Figure 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/or 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, due to 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 the 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. 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, 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 system throughput (STP) [7]), 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. Figure 3 shows SER reduction and performance loss for the SER-optimized sched-

ule 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% reduction in SER (and up to 62.8%) while degrading performance by 7% 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 (SER), 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.

Let us first recap the definition of soft error rate (SER) for single-program workloads:

$$SER = \frac{ABC}{T} \times IFR, \tag{1}$$

with ABC defined as the total ACE bit count over the entire execution of a program. 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:

$$wSER = \frac{ABC}{T} \cdot \frac{T}{T_{ref}} \cdot IFR = \frac{ABC}{T_{ref}} \cdot IFR, \tag{2}$$

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 duration.

Summing the weighted SER values for the individual applications in a multiprogram workload then yields our novel *system-level soft error rate (SSER)* metric:

$$SSER = \sum_{i=1}^{n} wSER_i = \sum_{i=1}^{n} \frac{ABC_i}{T_{i,ref}} \cdot IFR, \tag{3}$$

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

| **(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 |

**Table 1: Examples illustrating the SSER metric.**

## 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 — 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 — example (b) in Table 1. In this case, SSER equals 3, i.e., a weighted SER of 1 for the 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 two times as vulnerable.

Consider now a heterogeneous multicore — example (c) in Table 1. 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. 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 way 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 counter 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 each core type. If the number of big cores equals the number of small cores, this requires two sampling quanta: putting half of the applications on a big core and half on a small core in the first sampling quantum, and inverting 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.

```
if No sampling data available, or one or more applications have
run on the same core type for at least 10 scheduler quanta
then
    Enter sampling phase: run the application(s) on the core
    type for which sampling data is missing or stale for the next
    sampling quantum/quanta
else
    for all Applications do
        Calculate wSER reduction or increase if scheduled on
        other core type based on sampled data
    end for
    while Couples of applications exist where wSER reduction
    is larger than wSER increase when switched do
        Switch these applications for the next scheduler quantum
    end while
end if
Record performance and ABC for each application in the cur-
rent schedule
```

The algorithm first verifies whether the sampled data is recent. If an application has run for 10 consecutive scheduler quanta on the same core type, a sampling phase is triggered: the application is scheduled on the other core type by switching it (during a short sampling quantum) with the application that is running for the most consecutive quanta on the other core type. Like this, 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 switching the two applications leads to a net overall SSER reduction. If so, the applications are switched, and the next couple is checked. If no global SSER reduction can be obtained, the current schedule is selected for the next scheduler 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 (see Section 3). 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 in order 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

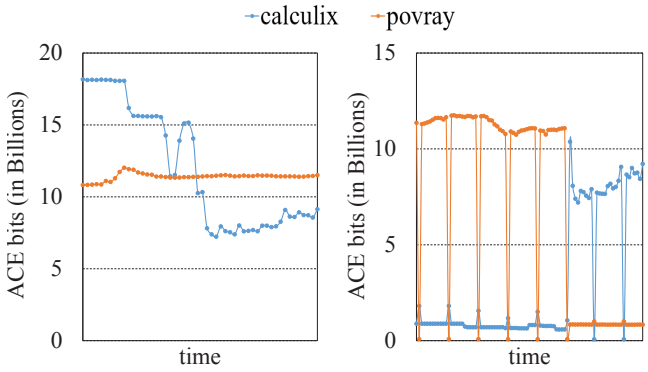


**Figure 4: ABC over time for `calculix` and `povray` when executed in isolation on a big core (on the left) and as a two-program workload on one big core and one small core under reliability-aware scheduling (on the right).**

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.

Figure 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 only 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`.

## 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 4096 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 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 occupation time in each of the five structures to the respective overall counters. 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 3072 bits for an 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 times 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 [27] 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 counter (ABC) stacks shown

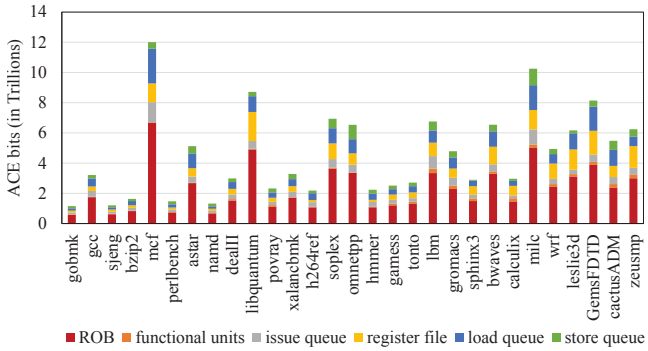


**Figure 5: ABC stacks for the out-of-order core.**

in Figure 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. 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.

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 (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 using simulation. We use Sniper 6.0 [4] using its most detailed cycle-level core model. 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 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. [18]). We do not include the cache in the ACE calculation, because the cache configuration is the same for both core types, and caches typically include error detection and correction mechanisms, making them less vulnerable to soft errors. Our default configuration assumes the same frequency for both core types, but we also evaluate the impact of having a lower frequency on the small core than on the big core.

| | *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)<br>1 int mult (3 cyc)<br>1 int div (18 cyc)<br>1 fp add (3 cyc)<br>1 fp mult (5 cyc)<br>1 fp div (6 cyc) | 2 int add (1 cyc)<br>1 int mult (3 cyc)<br>1 int div (18 cyc)<br>1 fp add (3 cyc)<br>1 fp mult (5 cyc)<br>1 fp div (6 cyc) |
| Register file | 120 int (64 bit)<br>96 fp (128 bit) | 16 int (64 bit)<br>16 fp (128 bit) |
| L1 I-cache | 32 KB, ass 4, 2 cyc | 32 KB, ass 4, 2 cyc |
| L1 D-cache | 32 KB, ass 8, 4 cyc | 32 KB, ass 8, 4 cyc |
| Private L2 cache | 256 KB, ass 8, 8 cyc | 256 KB, ass 8, 8 cyc |
| Shared L3 cache | 8 MB, ass 16, lat 30 cyc | |
| Memory | BW 25.6 GB/s, lat 45 ns | |

**Table 2: Big and small core configurations.**

The overhead of moving applications between cores is modeled as 20 $\mu$s per migration [8], for saving and restoring architectural state. This overhead has a negligible impact on the final results: less than 1% for a random scheduler that switches every quantum, and less than 0.5% for the reliability-optimized scheduler.

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 sensitivity to reliability-aware scheduling, see also Figure 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 — but we also make sure that each benchmark occurs at least once — leading to 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 two-program workloads on an HCMP consisting of 1 big and 1 small core (denoted 1B1S). The four-program workloads are evaluated on a symmetric

HCMP configuration consisting of 2 big and 2 small cores (2B2S), and also on asymmetric HCMP configurations: 1 big, 3 small cores (1B3S) and 3 big, 1 small cores (3B1S). The eight-program combinations are evaluated on a symmetric HCMP with 4 big and 4 small cores (4B4S). 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 its turn impacts the other applications. Taking results from the first execution only for the repeating applications would not cover these changes in the schedule.

# 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 from Section 4.
- The *performance-optimized scheduler* optimizes system throughput (STP) [7] or weighted speedup, using the same sampling-based scheduling algorithm optimizing for STP rather than SSER.

We first analyze the results for the 2B2S configuration. Next, we show how our scheduler performs for different core and application counts, as well as for asymmetric HCMP configurations. We also show the impact of using only ROB ACE bits to steer scheduling, the impact of the sampling period, and the impact of reducing the frequency of the small core.

## 6.1 2B2S Results

Figure 6 evaluates system soft error rate (SSER) and system throughput (STP) 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% on average and up to 55.6% compared to the random scheduler; and by 25.4% on average and by up to 60.2% 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% on average). This improvement is substantially smaller and, moreover, it

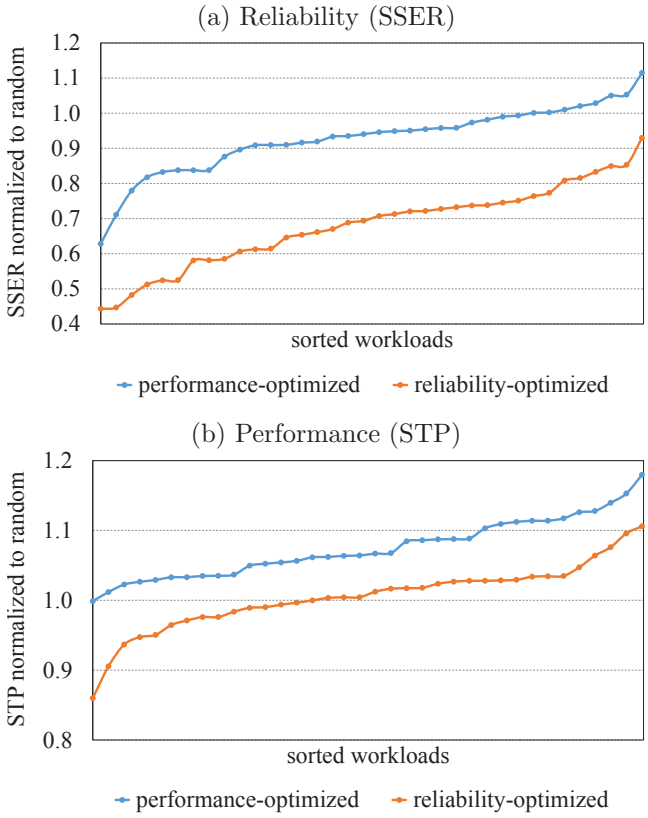


**Figure 6: System soft error rate (a) and system throughput (b) for reliability- and performance-optimized scheduling normalized to random scheduling for all four-program workloads on an HCMP with 2 big cores and 2 small cores.**

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% difference), and degrades performance by only 6.3% on average (and by 18.7% at most) compared to the performance-optimized scheduler. The performance improvement of performance-optimized scheduling over random scheduling is in line with prior work [28].

## 6.2 Analysis by Workload Category

Figure 7 shows the same results as Figure 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

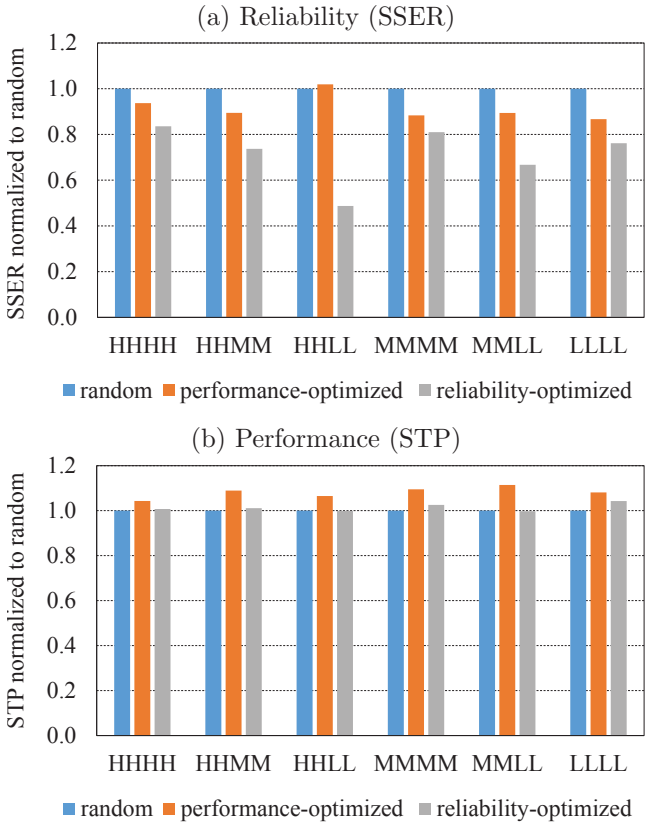


**Figure 7: SSER (a) and STP (b) on a 2B2S system per workload category.**

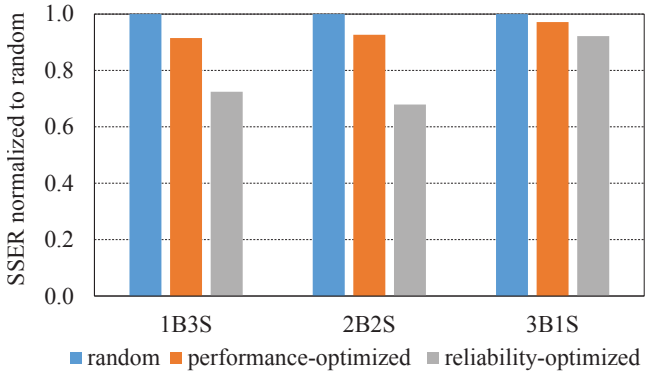


**Figure 8: SSER across asymmetric HCMPs with 4 cores in total.**

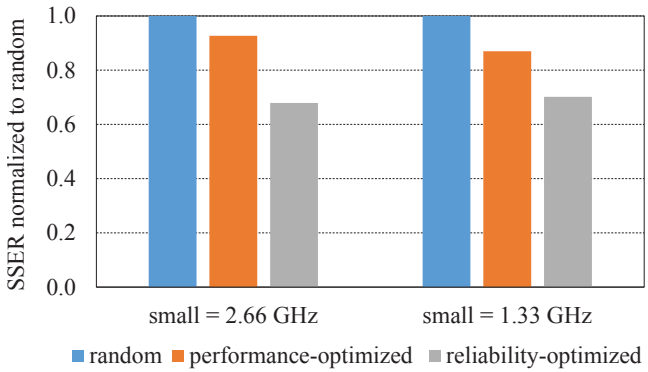


**Figure 9: SSER for the 2B2S system with the small cores running at different frequency settings.**

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 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.

## 6.3 Asymmetric HCMPs

The results in the previous sections assume symmetric HCMPs, i.e., the number of big cores equals the number of small cores. We now evaluate asymmetric HCMP configurations for four-program workloads: 1 big and 3 small cores (1B3S), and 3 big and 1 small cores (3B1S), see Figure 8. (Performance is within 7.8% of the performance-optimized scheduler across the three configurations.) The most noteworthy observation from this graph is that the highest reduction is obtained for the symmetric HCMP configuration; this is due to the fact that there are more scheduling opportunities on the symmetric HCMP than on the asymmetric HCMPs, i.e., 2 out of 4 applications need to be selected to run on a small core on the symmetric HCMP (2 combinations out of 4 leads to 6 possibilities) as opposed to one application to run on the big or small core in the asymmetric HCMPs (1 combination out of 4 leads to only 4 possibilities). The reduction in SSER on the 3B1S system (7.8%) is smaller than on the 1B3S system (27.5%) because there is only one small core available in the former system; this limits the scheduling opportunities to reduce soft error rate by migrating a program to the small core.

## 6.4 Lowering Small Core Frequency

So far, we assumed that the big and small cores run at the same frequency. We now evaluate the robustness of the reliability-aware scheduler with respect to frequency setting, see Figure 9. To this end, we set small core frequency to 1.33 GHz while running the big core at 2.66 GHz. The bottom line is that reliability-aware scheduling is robust with respect to frequency setting: system reliability improves by 29.8% compared to random scheduling for the low-frequency small core. This improvement is slightly smaller compared to the high-

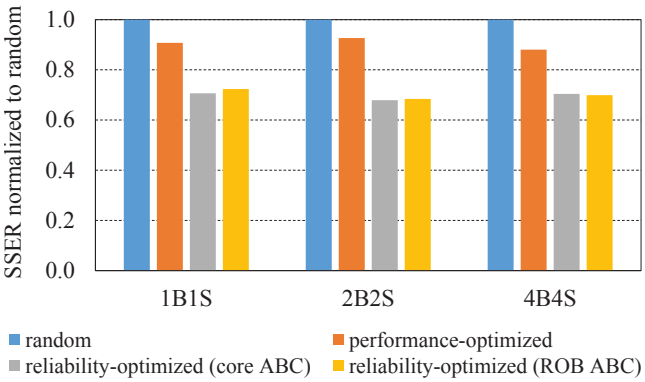


**Figure 10: SSER as a function of core count, assuming symmetric HCMPs and considering ROB ABC in addition to core ABC.**

frequency small core case because lowering the small core's frequency also lowers its performance, which increases its weighted SER because of the larger slowdown compared to big-core performance. This reduces the opportunity for reliability-aware scheduling to improve reliability compared to the high-frequency small core case. The performance-optimized scheduler on the other hand improves reliability more for the low-frequency small core than for the high-frequency small core (13% versus 7.3%) compared to random scheduling. This is a side-effect of the wider gap between big and small core performance. Performance-optimized scheduling improves overall system performance compared to random scheduling (by 10% on average), which decreases an application's weighted SER and improves overall system reliability more than random scheduling.

## 6.5 Changing Core Count

Figure 10 evaluates SSER across two-, four- and eight-program combinations on symmetric HCMPs (1B1S, 2B2S, and 4B4S). The results are consistent across core counts: the reliability-optimized scheduler significantly improves system soft error rate compared to random scheduling by 29.3%, 32% and 29.8% on average for 1B1S, 2B2S and 4B4S, respectively, while yielding comparable performance to the random scheduler, and only slightly worse performance compared to the performance-optimized scheduler (within 6.3% on average). This result shows that our scheduler scales well with core count and the number of co-running applications.

## 6.6 ROB ACE Bit Counter

Up to now, we assumed an ACE bit counter for all structures. To reduce hardware overhead by a factor of 3, as previously described in Section 4.2, the area-optimized implementation counts ACE bit information in the ROB only. Figure 10 shows SSER for reliability-aware scheduling using core ABC versus ROB ABC. The relative difference is negligible (31.6% for ROB ABC versus 32% for core ABC for the 2B2S system), which justifies the reduction in hardware cost by only tracking ROB ABC.

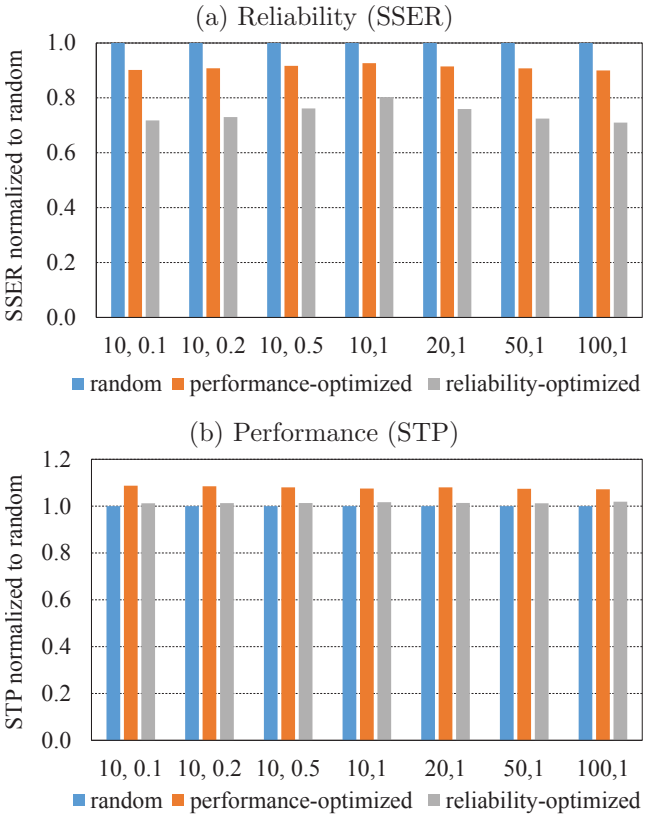


**Figure 11: SSER (a) and STP (b) for a 2B2S system while varying the sampling parameters $(r, s)$, i.e., sampling every $r$ quanta for $s$ milliseconds (i.e., the sampling quantum).**

## 6.7 Sample Rate

Figure 11 evaluates the impact on system reliability and performance while varying the sample rate to keep the big and small core performance and soft error rates up to date, see also Section 4.1. Our default sampling period SP is set to 10, i.e., we initiate a sampling phase every 10 scheduler quanta, and we sample for a sampling quantum of 0.1 milliseconds. Two interesting observations are to be made here. First, reliability improves for smaller sampling quanta as a result of reduced sampling overhead. Second, reliability improves as we increase the sampling period, i.e., as we sample less frequently. This suggests that our workloads show relatively stable time-varying execution behavior. However, some workloads clearly benefit from having a high sample frequency. For example, the workload consisting of xalancbmk, soplex, leslie3d and dealII has a 18.4% reduction in SSER for a sampling period of 10, whereas SSER reduces by only 10% for a sample period of 100.

## 6.8 Power Consumption

Changing the schedule in a heterogeneous multicore obviously affects power consumption. Figure 12 quantifies the impact on chip-level power (including L3) and total system power (processor plus DRAM). We use McPAT [12] to quantify power consumption. The bot-

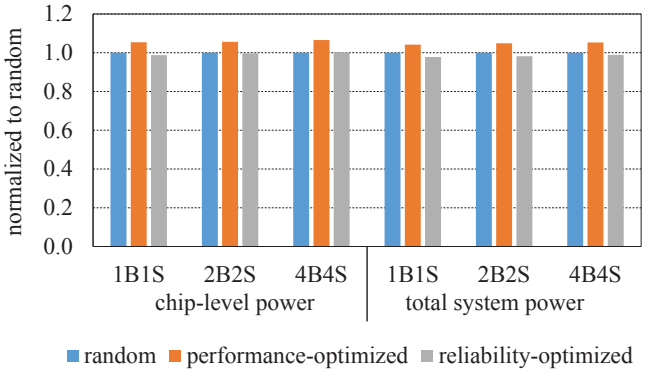


**Figure 12: Impact on chip-level and total system power consumption.**

tom line is that reliability-optimized scheduling reduces chip-level and system power by 6% and 6.2% 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 [28]; 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. RELATED WORK

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

## 7.1 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 [3], error detection and correction mechanisms [20], or architectural techniques [30, 25]. 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 [13]. Mukherjee et al. [16] 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. [2] show how to measure AVF for address-based structures. Sridharan and Kaeli [26] 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 [29, 14], or through analytical mechanistic modeling [18]. Nair et al. [19] 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.

## 7.2 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 [5, 9, 22]. Van Craeynest et al. [28] 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 [15] or power efficiency [17, 31]. We are the first to improve reliability on HCMPs through scheduling.

# 8. 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 (SSER), 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% on average and up to 60.2% compared to performance-optimized scheduling, while degrading performance by 6.3% only. The proposed scheduler is robust across core count, number of big versus small cores, and frequency settings. Moreover, as a side effect, reliability-aware scheduling reduces power consumption by 6.2% on average compared to performance-optimized scheduling.


## Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295.

# 9. REFERENCES


[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[2] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA)*, pages 532–543, 2005.

[3] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, pages 2874–2878, 1996.

[4] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28, 2014.

[5] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 927–930, 2009.

[6] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the High Performance Computer Architecture (HPCA)*, pages 1–8, 2012.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[8] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, September 2011.

[9] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, pages 125–138, 2010.

[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 *36th International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[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 *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 64–75, 2004.

[12] 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 *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[13] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 341–352, 2008.

[14] D. Lide, L. Bin, and P. Lu. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 129–140, 2009.

[15] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 317–328, 2012.

[16] 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 *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.

[17] T. S. Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–176, 2014.

[18] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John. A first-order mechanistic model for architectural vulnerability factor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 273–284, 2012.

[19] 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 *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*, pages 125–136, 2010.

[20] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, 2005.

[21] NVidia. Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf, 2011.

[22] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, 2002.

[24] 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 *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[25] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 506–515, 2007.

[26] V. Sridharan and D. R. Kaeli. Using hardware vulnerability factors to enhance AVF analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 461–472, 2010.

[27] R. Uma, V. Vijayan, M. Mohanapriya, and S. Paul. Area, delay and power comparison of adder topologies. *International Journal of VLSI design & Communication Systems (VLSICS)*, 3(1), 2012.

[28] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224, 2012.

[29] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 516–527, 2007.

[30] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 264–275, 2004.

[31] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, 2015.