# Parallaft: Runtime-Based CPU Fault Tolerance via Heterogeneous Parallelism

Boyue Zhang
University of Cambridge
Cambridge, United Kingdom
bz275@cl.cam.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, United Kingdom
sam.ainsworth@ed.ac.uk

Lev Mukhanov
Queen Mary University of London
London, United Kingdom
l.mukhanov@qmul.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, United Kingdom
timothy.jones@cl.cam.ac.uk

## Abstract

The increasing vulnerability of microprocessors due to frequent silicon faults greatly exacerbates the risks of silent data corruption. Existing software-based schemes to detect these suffer from high power and performance overhead. State-of-the-art hardware fault-tolerance techniques exploit processor heterogeneity to minimize power, performance, and area overhead, but have not seen deployment in production due to their complexity.

This paper shows for the first time that the same insights of heterogeneous parallelism can be repurposed without any hardware support. We present Parallaft, a parallel software-based error detection technique taking the insights of state-of-the-art hardware techniques and repurposing them with tools more suited to the hardware of today, such as copy-on-write checkpointing, dirty-page tracking and performance-counter synchronization. This allows error checking to be offloaded to little cores of an Apple M2 heterogeneous processor, achieving half energy cost while maintaining performance comparable to the homogeneous duplication mechanism RAFT.

*CCS Concepts:* • **Software and its engineering** → **Software reliability**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

***Keywords:*** Software reliability, Heterogeneous processors, Runtime

**ACM Reference Format:**
Boyue Zhang, Sam Ainsworth, Lev Mukhanov, and Timothy M. Jones. 2025. Parallaft: Runtime-Based CPU Fault Tolerance via Heterogeneous Parallelism. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708946

## 1 Introduction

The semiconductor industry faces the risks of transient and permanent faults in processors with silicon miniaturization, growth in transistor numbers, and voltage reduction [14, 42, 46, 57], despite existing reliability, availability, and serviceability (RAS) [18, 36] features such as error correction code (ECC) [11] memory and parity-checked network-on-chip communication [3]. Beyond causing detectable symptoms such as exceptions, these faults may even lead to silent data corruption (SDC). Data-center operators have observed that SDCs cause severe issues at scale [24, 26], since the manifestation of a permanent fault may be delayed until some time later in the processor lifetime, may only occur at specific temperature, frequency and voltage conditions and may only affect certain instructions [24]. These have become common on desktop-class systems as well [16].

In safety-critical industries such as automotive and healthcare, custom hardware-based lockstep systems [28] are used to protect from both transient (soft) and permanent (hard) errors. In these systems, copies of a program execute on multiple processors, enabling error detection when the outputs of the synchronized processors differ. However, most commercial processors lack this capability. To address this limitation, software-based schemes have been developed to replicate execution on commodity processors [29, 43, 55], eliminating the need for hardware modifications. However, both hardware-based and software-based techniques lead to a doubling in energy consumption, and either double silicon area or halve performance to perform the repeat run. These factors severely limit the adoption of such techniques in performance-critical environments. Hence, to detect faulty processor cores with potential hard or semi-hard errors, software scanners are used at scale [13, 23], which periodically run tests on processors looking for errors. However, they either only provide limited error coverage or force servers out-of-production for long periods.

With the rise of heterogeneous hardware, a new class of fault tolerance techniques, such as ParaMedic [8–10], has been proposed to lower power, performance and area overhead. Their insight is that the second, error-checking run can be parallelized to run on a sea of small, power-efficient cores while keeping up with the main execution on a big core. Despite these insights, none have yet seen commercial deployment due to the required hardware complexity.

To address the deployability issues of these techniques while reusing the same insights, this paper presents Parallaft, a heterogeneous parallel error-detection runtime system. Parallaft exploits ParaMedic-style parallelism to take advantage of heterogeneous processors to minimize power overhead, but without the need of hardware modification or program recompilation. Parallaft is implemented as a user-space program that takes a binary executable, slices its execution into segments, and duplicates execution of each segment to compare results. To do this, we translate prior work's induction parallelism principle [8] into a new collection of software primitives, including copy-on-write checkpoints, execution point record-and-replay capability, optimized program-state checking, and energy-efficient scheduling.

This paper makes the following contributions:

- We present Parallaft, a runtime heterogeneous parallel error-detection technique [8, 9] that splits execution into multiple segments, replaying each segment on one of several little cores to check results.
- We demonstrate that Parallaft incurs half energy overhead while only introducing comparable performance overhead when compared with the previous state-of-the-art runtime fault tolerance solution, RAFT [55].
- We analyze the contribution of different factors to Parallaft's overhead, such as resource contention and runtime work, and performance tradeoffs. Finally, we verify the efficacy of Parallaft via fault injection.

## 2 Motivation and Related Work

Table 1 presents a variety of hardware and software techniques used to duplicate execution for fault tolerance. Here we discuss how we take insights from each in order to build an energy-efficient pure-software solution.

### 2.1 Traditional (Homogeneous) Error Detection

Traditionally, to protect safety-critical systems from soft- or hard-errors, lockstep systems [28, 44, 52] are used, where two or more identical processors are synchronized to run the same computation. An error is flagged when the computation results mismatch. With three or more synchronized processors, it is possible to correct errors by majority voting [28]. Another approach to redundancy avoids strict synchronization: one processor runs ahead of the other, speculatively skipping some instructions. This method offers partial redundancy while also boosting performance [47].

Since hardware schemes require custom processors, to address deployability issues, software-based schemes have been proposed. These techniques can be broadly classified into compiler- [19–22, 33, 41, 45, 56] or runtime-based [43, 55] approaches. In compiler-based schemes, to protect programs from errors, additional instructions are inserted into the original program by the compiler to duplicate computation and check results. However, these techniques require recompilation of the source code, which limits applicability. In runtime-based techniques, a runtime duplicates the program execution dynamically (e.g. through a different process) and checks if the behavior of two executions diverges. For example, RAFT [55] intercepts interactions (e.g. syscalls and signals) between the operating system and the program. Rather than requiring exact lockstep coupling [28, 44, 52], it allows the two computation threads to run asynchronously on standard compute cores, with comparisons performed only on external output such as syscalls (figure 1(a)). It extends PLR [43] to improve the performance of syscalls by speculating on them rather than waiting for synchronization between the two redundant copies.

However, the techniques described above need the duplicate execution to run on another processor core of equivalent performance to the original run to avoid slowdown [28, 44, 52, 55], or interleaved with the original execution on the same core [20–22, 33, 35, 49]. This naïve duplication means that we double the energy to provide reliability.

### 2.2 Heterogeneous Parallel Error Detection

To reduce energy cost, heterogeneous parallel error detection techniques, such as ParaMedic, have been proposed [8–10]. Their insight is that the second, error-checking run is more parallel than the original run. The parallelism enables error-checking on a sea of smaller cores, by splitting the program execution into segments and overlapping checking of multiple segments. Though each individual little core has less computational power than a big core, with the parallelism, all little cores together can provide enough computation to keep up with the original execution (figure 1(b)).

These schemes work by recording, replaying and comparing program execution at the architectural level. During the main execution, all loads and stores are recorded in a hardware load/store log, which are then replayed to the checkers. In addition, register checkpoints are taken at the beginning and the end of each segment. An error is flagged when a checker's behavior diverges from the load/store log (e.g. an attempt at loading from an incorrect address) or when the checker registers do not match with the register checkpoint at the end of the segment.

Other works also use heterogeneity in more diverse forms. DIVA [12] observed that instruction-level parallelism can be used to execute a verification run on a high-ILP in-order superscalar core. Argus [31] instead exploits heterogeneity by task-splitting for the redundant run, handling control,

**Table 1.** Comparison among processor fault-tolerance techniques.

| Approach | | State-of-the-art techniques | Needs specialized hardware | Needs source code | Memory overhead | Performance overhead | Energy overhead |
|---|---|---|---|---|---|---|---|
| Hardware-based | Lock-stepping | TCLS [28], IBM [44], Cortex-R [52] | Y | N | 0 | ~0 | ~100% |
| | Simultaneous multithreading | RMT [35], SRTR [49] | Y | N | 0 | 32% (RMT [35]), 60% (SRTR [49]) | ~100% |
| | Parallel heterogeneous | ParaMedic [9] | Y | N | 0 | 3% (ParaMedic [9]) | 16% |
| Compiler-based | Thread-local duplication | SWIFT [41], nZDC [20], mZDC [21], gZDC [22], InCheck [19] | N | Y | ~0 | 45% (SWIFT [41]), 197% (InCheck [19]) | ~100% |
| | Redundant multi-threading | DAFT [56], COMET [33], EXPERT [45] | N | Y | ~0 | 38% (DAFT [56]), 400% (EXPERT [45]) | ~100% |
| Runtime-based | Asynchronous duplication | PLR [43], RAFT [55] | N | N | 95% | 16.2% | 87.8% |
| | Parallel heterogeneous | Parallaft (This paper) | N | N | 232% | 15.9% | 44.3% |

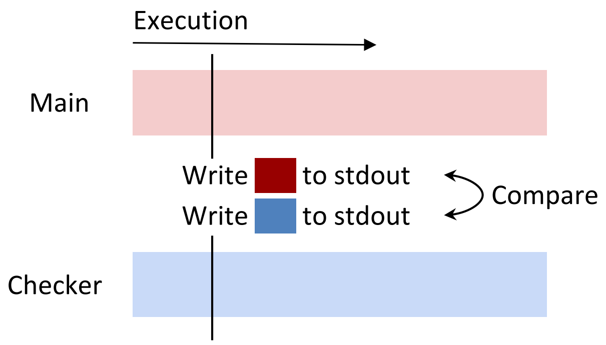
data-flow, computation and memory as separate verification tasks. We focus on ParaMedic-style [8, 9] thread-level heterogeneous parallelism because it is less clear how to translate the insights of these other mechanisms into commodity heterogeneous cores.

The main drawback of these schemes is that they require hardware modification, limiting real-world deployment.

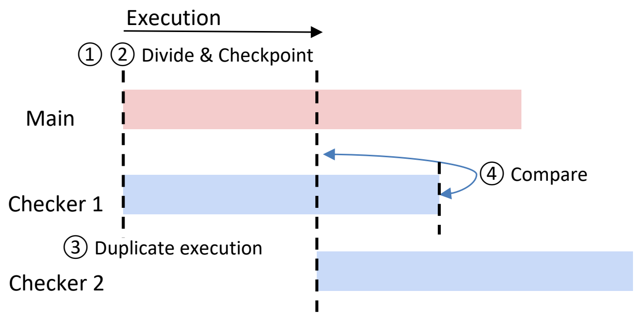
### 2.3 From RAFT to Parallaft

Figure 1(a) shows a simplified example of RAFT [55]. RAFT is a compelling baseline because a) it is relatively high performance compared to other software techniques (table 1), b) requires no recompilation of binaries, and c) spreads the main computation and the trailing redundant computation between two cores, exploiting parallelism compared to other software techniques [20, 41]. With Parallaft, we want to go much further: rather than executing the redundant computation on one core (which must be as fast as the original execution), we want to split the second, checker copy across several more energy-efficient, slower cores, just as in ParaMedic [8–10], but without any hardware support. We thus split the checker process into *multiple different segments* (figure 1(b)), and must reproduce the computation deterministically until each segment ends, comparing the results with the start of the next checkpoint to demonstrate correctness of the entire program via induction [8].

Migrating the ParaMedic strategy [9] to software without incurring excessive performance cost presents numerous challenges. To enable execution record and replay, ParaMedic logs each executed load and store operation in a segment combined with a register checkpoint [9]. However, in software, logging each load and store is very costly [33], especially if recompilation of the source code is not possible [39]. Our insight is to use virtual-memory-based copy-on-write checkpoints combined with deterministic record-and-replay [15, 37] (section 3.2). To check for errors in the checker run, instead of comparing each load and store, Parallaft tracks modified pages [1] and compares the hash of the data inside these pages after a checker finishes its segment (section 4.4). Moreover, to record the execution point



**(a)** Execution plan of RAFT, where the original main process and a duplicated checker process run asynchronously on two processor cores, with their interactions with the OS, such as syscalls (e.g. `write` in the diagram), intercepted and compared.



**(b)** Execution plan of Parallaft, where the original main execution is ① divided into segments. At the boundary of each segment, ② a checkpoint is taken. Each segment is then ③ executed by a checker independently of other checkers, allowing parallelism. At the end of each segment, ④ the state between the segment-end checkpoint and the checker is compared to flag any faults.

**Figure 1.** From RAFT to Parallaft.

when the segment ends, ParaMedic [8, 9] counts the number of instructions executed in the segment. Intuitively, one might expect to be able to do the same with existing hardware performance counters. However, in commodity processors, precisely counting instructions without code instrumentation is nearly impossible due to nondeterminism and overcounting on modern processor hardware performance counter implementations [51]. Instead, we use branch counters[1] combined with breakpoints (section 4.2).

While splitting a task into different segments and executing each segment in parallel is not new in profiling techniques [30, 34, 53], they neither ensure faithful replay of segments nor guarantee that consecutive segments are linked together without gaps or overlaps, as neither is strictly necessary to get useful profiling results. In contrast, they are essential in Parallaft to avoid false-positive detection, which brings new challenges for record and replay.

## 3 Parallaft: Design Overview

Parallaft acts as a layer between the application and the operating system (OS), as shown in figure 2. Under the tracing of Parallaft, the main process executes the program while the checkers on little cores verify correctness by running segments of the main program in parallel. A *coordinator unit* residing in the Parallaft process manages checkpoints, segments, program slicing, execution points, and record-and-replay mechanisms. System interactions like syscalls and signals, and nondeterministic instructions are recorded and replayed to maintain consistency. At the end of each segment, Parallaft compares hashes of data inside modified memory pages using a *dirty-page tracker* and a *program-state comparator.* Additionally, under the control of checker scheduler and pacer, if little cores fall behind, checkers are moved to a larger core. Conversely, if the checker cores collectively have more computation power than needed to keep up, their frequencies are lowered for better energy efficiency.

### 3.1 Heterogeneous Parallelism

To achieve energy efficiency, Parallaft exploits parallelism in error checking, hence allowing checkers to run on multiple little cores in a heterogeneous processor (section 2.2). Specifically, Parallaft slices the execution of the main program into multiple *segments.* Each segment is executed twice, once by the main and once by a checker. As the main program generates these segments from a sequential underlying process[2], it must finish one before generating the next. However, the execution of trailing segments can be overlapped. At a segment boundary (based on reaching a target number of instructions or cycles), Parallaft takes a copy-on-write checkpoint of the state of the main program, as

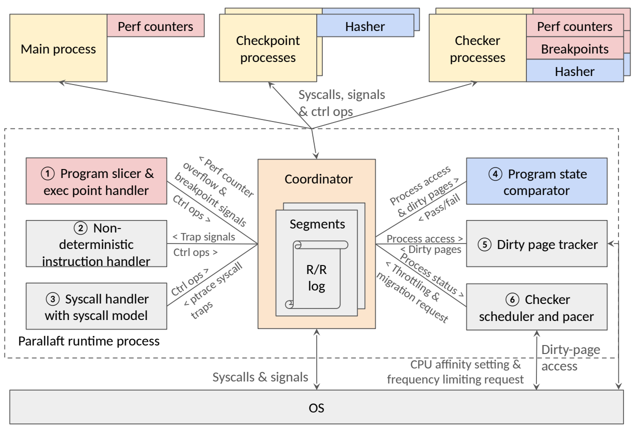

Main process
Perf counters
Checkpoint processes
Hasher
Checker processes
Perf counters
Breakpoints
Hasher
Syscalls, signals & ctrl ops
① Program slicer & exec point handler
< Perf counter overflow & breakpoint signals
Ctrl ops >
② Non-deterministic instruction handler
< Trap signals
Ctrl ops >
③ Syscall handler with syscall model
Ctrl ops >
< ptrace syscall traps
Coordinator
Segments
R/R log
Process access & dirty pages >
< Pass/fail
④ Program state comparator
Process access >
< Dirty pages
⑤ Dirty page tracker
Process status >
< Throttling & migration request
⑥ Checker scheduler and pacer
Parallaft runtime process
Syscalls & signals
CPU affinity setting & frequency limiting request
Dirty-page access
OS

**Figure 2.** System design of Parallaft. Parallaft acts as a layer between the application under protection and the OS. It traces the main application process and the derived checker and checkpoint processes (in yellow). A coordinator (in orange) handles interactions, such as syscalls, signals and control operations, with the processes. The ① program slicer first divides the main execution into multiple segments. For each segment, a checker and checkpoint processes are created by the coordinator. To maintain consistency between the main and the checker, the coordinator keeps a record and replay (R/R) log and talks with relevant event handlers, such as the ① execution point handler, the ② nondeterministic instruction handler, and the ③ syscall handler, to record and replay the events. To enable execution point record and replay, the ① execution point handler attaches hardware performance counters and breakpoints (in red) to the main and the checker processes. In addition, to compare the state at the end of each segment, the ⑤ dirty page tracker tracks memory pages modified and passes the dirty pages to the ④ program-state comparator via the coordinator. The ④ program-state comparator compares the hash of data in modified pages via a injected hasher code (in blue), in addition to registers. Moreover, there is the ⑥ checker scheduler and pacer to migrate checkers to big cores and adjust checker-core frequencies.

well as forking a checker from the main process, such that the checker starts from the same state as that of the main. Each checker executes in its own address space and registers, independently of the main process and other checkers. When a checker finishes its execution to the end of its segment, the checker state (i.e. memory and registers) is compared against the next checkpoint, with any miscomparsion flagged as errors.

To keep up with the main execution, multiple checkers on a number of little cores execute concurrently, such that their combined computational power is enough to keep up. Thanks to the principle of induction [38], which ensures correctness for the whole program by proving correctness for

[1]Or in some microarchitectures, a combination of branch counters to workaround the overcounting issue (section 4.2.1).

[2]Parallaft's prototype, just like RAFT [55], does not yet support multi-threaded applications, because recording and replaying nondeterministic shared-memory operations in software is complex. Hardware techniques solve this problem by employing a hardware load-store log [9] or monitoring cache coherence traffic [50]. However, they are difficult to translate into software without high-overhead binary translation. Instead, we plan to explore the use of DoublePlay-like functionality [48] to allow Parallaft to extend to such a domain, via speculation on races and rollback.

**Table 2.** Error containment, detection, and recovery capabilities of RAFT and Parallaft.

| | **RAFT** | **Parallaft** |
|---|---|---|
| Guaranteed error detection | No | Yes |
| Error containment in SoR | No | Future work |
| Error recovery possible? | No | Future work |

individual segments and their connections, the entire execution is correct as long as all segments are correct.

### 3.2 Execution Duplication

To maintain consistency between the main and the checker executions and to ensure syscall copies appear only once to the outside world, Parallaft intercepts and records application/OS interactions (e.g. syscalls and signals) from the main execution, and replays the effects of the interactions to the checkers after ensuring they are correct. Parallaft also traps, records and replays any nondeterminism, such as x86_64 `rdtsc` instructions and AArch64 `mrs` instructions.

In addition, Parallaft records and replays execution points, to allow checkers to stop at an execution point where the main finishes the segment. To achieve this, Parallaft uses hardware branch counters with breakpoints, which is similar to techniques used in RR debugger [37] and ReVirt [25].

### 3.3 Process-State Comparison

To speed up state comparison between the main and the checker at the end of each segment, Parallaft tracks and compares modified memory pages during the segment. To minimize memory copying, Parallaft injects the target process with code that computes a hash of the modified pages, and compares the hashes; registers are compared as well.

### 3.4 Sphere of Replication

In Parallaft, the sphere of replication (SoR) [40], or components protected from faults by our technique, is the user-space execution of the program binary under protection, including dynamically loaded libraries and dynamically generated code. Excluded from our SoR are the operating system and our reliability runtime system, but most applications spend the vast majority of their time in user-mode, and so it is possible to protect these components using compiler-based fault tolerance techniques such as gZDC [22] without large overheads. Moreover, we assume the hardware memory subsystem is protected by ECC or parity.

While PLR [43] and RAFT [55] intend that data (such as syscalls) escaping out of the sphere-of-replication is correct, our Parallaft prototype does not make this guarantee to avoid synchronization overheads, which are more challenging in our environment when the checker runs are executed on slower, more energy-efficient cores. Instead, Parallaft eagerly passes main-issued syscalls to the OS before comparing them against those issued by the checker, meaning that though all errors will be detected, they are allowed to propagate to other processes before this detection point. This is the behavior in hardware equivalents [8]. We assume buffer layers around processes and non-reversible IO in order to undo any subsequently discovered damage. However, unlike RAFT, where syscall misspeculation may let erroneous data escape and not be detected later[3], Parallaft does guarantee all errors are detected within a configurable latency upper limit, i.e. the maximum single segment length times maximum number of live segments. In our experiments, we compare like-for-like, in that RAFT also pays no penalty for syscall synchronization.

Table 2 shows a summary of the error containment, detection, and recovery capabilities of RAFT and Parallaft. As discussed, Parallaft guarantees all errors are detected by its design, but does not guarantee error containment in the SoR to avoid costly syscall synchronization overhead. It is future work to prevent errors escaping without introducing excessive overhead and to add rollback-based error recovery. In comparison, due a design bug in RAFT's syscall misspeculation recovery (as discussed in footnote 3), RAFT neither detects all errors nor guarantees error containment in the SoR, hence making error recovery impossible.

## 4 Parallaft: Implementation

We implement Parallaft's prototype as a Linux user-space program in Rust, supporting x86_64 and AArch64. Parallaft slices the application under protection into multiple segments, so that checkers running on multiple little cores can run concurrently (section 4.1). To maintain consistency between the main and duplicated checker execution, Parallaft records and replays execution points at segment ends (section 4.2), syscalls (section 4.3.1 and section 4.3.2), signals (section 4.3.3), and nondeterministic instructions (section 4.3.4). To check for correctness, Parallaft compares registers and modified memory at the end of each segment (section 4.4). Moreover, Parallaft employs migration and dynamic frequency scaling techniques to schedule and to pace the checkers to gain maximum energy efficiency while allowing them to keep up with the main execution (section 4.5).

[3]Parallaft's periodic checkpoint matching, rather than RAFT's speculation and comparison only at syscalls, fixes a correctness bug in RAFT's intended policy. RAFT only detects an error on a syscall mismatch, meaning errors can exist hidden in state for long periods before manifesting. RAFT also speculates syscall results to let the copies run far ahead from each other asynchronously. In case of misspeculation, RAFT reverts the speculative process to the state of the non-speculative process. However, if the non-speculative process contains an error, it will never be detected. Parallaft avoids this issue because errors cannot escape beyond the bounds of the start and end of checkpoints, since all program state is compared at this point to allow the induction checks [8] to proceed.

### 4.1 Program Slicing

To exploit parallelism, Parallaft slices the program periodically into a number of segments to allow each to be checked concurrently with the others (figure 1(b)), based on the number of CPU cycles or instructions executed. We then combine branch-counting followed by a PC-based trap to reach the same instruction count on the replay run (section 4.2).

Unless otherwise noted, we use a slicing period of 5 billion cycles, meaning the main process will never run more than 5 billion cycles in user-space before the next checkpoint is taken. In our experimental platform, an Apple M2 processor, each big core can run at a frequency of up to 3.5 GHz, so a 5-billion-cycle segment translates to a user CPU time of around 1.43 s. The choice of 5 billion cycles is a trade-off: taking checkpoints too frequently results in an excessive number of forks and page copying, while taking checkpoints too infrequently results in the runtime waiting for the last checkers to finish after the main finishes[4].

### 4.2 Execution-Point Record-and-Replay

At each checkpoint, Parallaft records the execution point where the main process stops, and replays it by arranging the checker run to stop precisely at the same point. To do so, Parallaft utilizes processor hardware performance counters combined with breakpoints [4].

**4.2.1 Record Phase.** In the record phase, at each checkpoint, Parallaft sets up a hardware performance counter to count the number of branches retired in user-mode during the execution of the main process in the current segment. When the segment is sliced (section 4.1), Parallaft reads out the branch count and the current program counter (PC) value of the main process[5].

Parallaft relies on a deterministic and accurate count of branches in user-mode from hardware performance counters. Theoretically, recording and replaying the number of instructions is also possible. However, in practice, performance counters in most commodity processors overcount instructions nondeterministically from various sources, such as number of interrupt returns on Intel [51].

On Intel, the all-branch-retired counter suffers from nondeterminism where it overcounts by the number of returns from interrupts or exceptions [51]. To compensate, we exclude far branches, which include switches to a different privilege level plus intra-privilege level segment switches.

**4.2.2 Replay Phase.** In the replay phase, Parallaft replays the checker execution to the correct PC and branch count

[4]Given the benchmarks finish in around 120 s on average, a slicing period of 5 billion cycles gives less than 3% last-checker synchronization overhead. If Parallaft is used in an environment with short-running workloads, this should be reduced accordingly.

[5]A PC alone is not sufficient to reproduce the execution because it may be in a loop [32]. With the addition of the number of branches, we can replay to the correct iteration in the loop.

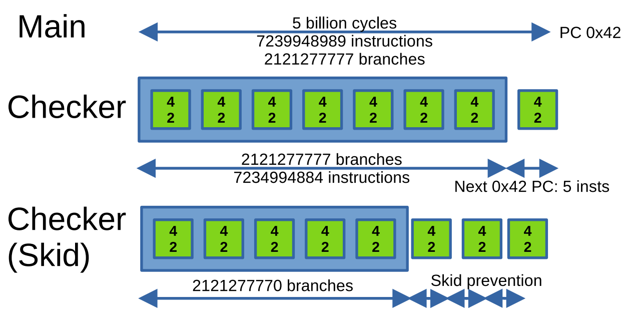


**Figure 3.** Example of how Parallaft reaches the same execution point on checkers as on the main core. After 5-billion cycles, the number of branches and PC is recorded. This is enough information for the checker to reach the identical program point by trapping at the number of branches (in the blue region), then setting up a breakpoint at the recorded PC. In the presence of performance-counter skid, we set a smaller threshold for the branch counter, breakpointing on the same PC many times until we reach our precise target.

(figure 3). Ideally, Parallaft simply 1) sets up the performance counter to stop checker execution after the recorded branch count, 2) after the desired number of branches are executed, sets up a breakpoint at the recorded PC, 3) on hitting the breakpoint, assert that the branch count stays the same, and the checker will be at the desired stop point.

However, in many commodity processors such as Intel, there is latency between receiving an overflow interrupt and the retirement of the instruction causing overflow, known as *skid*. Even with Intel precise events (PEBS [27]), we still see occasional skid in experiments on a Golden-Cove machine.

***Handling Skid.*** Skid causes checkers to over-run the desired stop point. To overcome this, we introduce a buffer: Parallaft sets the performance counter to overflow after most, but not all, branches we want the checker to execute. After an overflow, Parallaft enables the breakpoint as usual. Upon hitting the breakpoint, it compares the branch count with the desired value and continues execution unless the branch count equals to our desired value[6].

***Handling Timeout.*** Errors may cause control flow violation, which in turn, may render a checker never reaching the target program counter, causing infinite hang. To solve this problem, in each segment, Parallaft kills the checker when it executes more instructions than the instruction count of the main execution multiplied by a scale, currently set to 1.1. As

[6]Even with the introduction of the skid buffer, it is technically still possible that the checker overruns the end of the buffer. As we have not observed such cases on our evaluated microarchitectures, the current prototype of Parallaft does not handle this. However, it is possible to extend Parallaft to handle such cases by multiplicatively increasing the buffer length, and restarting the checker from the beginning of the segment.

instructions are counted using hardware performance counters, the scale is necessary to work around performance-counter overcounting and nondeterminism issues [51].

## 4.3 Execution Duplication

At the beginning of each segment, Parallaft starts replicating the execution of the main program by forking it to create a checkpoint and a *checker* process. The checker executes in its own address space, a copy-on-write copy of the main process's. Interactions between the main process and the operating system, such as memory-mapped IO (section 4.3.2), and syscalls and signals (section 4.3.3), are recorded and replayed to the checkers. Nondeterministic instructions, such as reads of timestamp counters (`rdtsc`) on x86_64, are also emulated, recorded and replayed to avoid divergence (section 4.3.4). In addition, Parallaft removes other nondeterminism, such as restartable sequences and vDSO, by masking these functionalities (section 4.3.5).

**4.3.1 Syscalls.** Parallaft intercepts and records syscalls from the main process with `ptrace` [5]. Parallaft keeps a *model* of each supported syscall [37], specifying which memory regions might be read or written given the syscall arguments. The model allows Parallaft to check whether the main and the checker processes make the exact same syscall, including any associated data (e.g. the data to write in a `write` syscall), as well as to replay the effect of the syscall on the process memory for the checker processes.

Parallaft handles different types of syscalls in different ways, which we categorize as follows [55].

***Globally-Effectful Syscalls.*** Syscalls with effects outside the sphere of replication, e.g. input-output (IO) syscalls like `write`. When the main process executes these syscalls, Parallaft records syscall inputs and outputs before and after passing the syscalls to the OS, respectively. In the future, when a checker executes these, Parallaft looks up the corresponding recorded syscall, checks the syscall inputs, and replays the outputs without passing them to the OS to avoid duplicated effect (e.g. writing the same data twice to a file).

***Process-Locally-Effectful Syscalls.*** Syscalls that affect the local state of the process, for example, process properties and memory mappings: e.g. `prctl`, `brk`, `mmap`, and `mprotect`. Parallaft passes most of them through to the OS in both main and checker processes, with additional handling of memory-related syscalls (section 4.3.2).

***Non-Effectful Syscalls.*** Syscalls that do not have external effects, but their outputs are usually nondeterministic or inconsistent between main and checkers: e.g. `gettimeofday` and `getpid`. Parallaft handles these syscalls the same way as effectful syscalls via recording and replaying.

**4.3.2 Memory-Mapped IO.** To handle memory-mapped IO, Parallaft records, replays and transforms `mmap` syscalls if necessary as follows.

***Address Space Layout Randomization (ASLR).*** When ASLR is enabled, the OS allocates a random address for each `mmap` syscalls, unless an address is explicitly asked by the caller. When the application makes an `mmap` syscall without explicit asking for a fixed address, ASLR leads to divergence in address-space layout. To fix this without requiring ASLR to be disabled, Parallaft records the address returned by the kernel for the main process. When the checker executes the same syscall, Parallaft modifies the `mmap` call, fixing the address to the recorded one with the `MAP_FIXED` flag. This is a similar technique to that used in RAFT [55].

***Private Mappings.*** Parallaft handles anonymous private `mmaps` as ordinary process-locally-effectful syscalls after fixing ASLR. For file-backed private mappings, Parallaft ends the current segment prior to the `mmap` call and starts a new segment after the call, placing the `mmap` call outside the protection zone to duplicate the memory map to the checker. Otherwise, the trailing checker call of `mmap` would fail as the file descriptor is invalid in checkers.

***Shared Mappings.*** Shared mappings are not used by the benchmarks we use for evaluation, and so we do not support them currently. Previous work [37] also avoids supporting them due to the complexity and overhead of tracking modifications that can occur from outside without warning, though it is technically possible using page permissions to trap every read access to these mappings, and recording and replaying the data read. We leave this to future work.

**4.3.3 Signals.** Parallaft traps each signal delivered to the application using `ptrace`. Like RAFT [55], on reception of a signal, Parallaft handles it differently based on its source.

***Internal Signals.*** If the signal comes from the application itself (e.g. `SIGSEGV`), Parallaft forwards the signal to the application and records the signal. Later, signals caught by checkers are checked against the record.

***External Signals.*** If the signal comes from outside the application (e.g. `SIGINT` when the user presses `Ctrl+C`), to avoid divergence, Parallaft delivers the signal at the identical execution point for both the main and the checker processes, in case the application has a custom signal handler[7]. To do this, Parallaft records the execution point of the main process at the time of signal delivery and then arranges the checker to stop at the same execution point before delivering the same signal to the checker.

**4.3.4 Nondeterministic Instructions.** Parallaft intercepts and emulates nondeterministic instructions to prevent

[7]RAFT does not support custom signal handlers, as it does not have execution point record and replay capability.

divergence. On x86_64, `rdtsc` (read time-stamp counter) and `cpuid` (CPU identification) instructions are disabled via hardware support, such that attempts to execute these instructions generate exceptions, which are then handled by Parallaft via emulate-record-and-replay[8].

On AArch64, Parallaft intercepts and emulates `mrs` (read system register) instructions. On a heterogeneous system, the same system register may have different values if read from different CPU cores. For example, one such register, `MIDR_EL1`, contains the core model. Parallaft traps these instructions by scanning the entire executable address space and replacing found `mrs` instructions with breakpoints[9], so later attempts to execute these instructions are handled via emulate-record-and-replay.

#### 4.3.5 Other Sources of Nondeterminism.

Nondeterminism can also arise from restartable sequences (`rseq`) [2] and vDSO [17]. Rseq refers to a special instruction sequence in a program that executes an abort handler when interrupted. While rseq enables certain optimizations, nondeterministic preemption can alter the program's control flow. Similarly, vDSO [17] is a library, potentially combined with shared memory, injected into the program's address space by the kernel to speed up specific syscalls, such as `gettimeofday`. In this case, the vDSO implementation reads timestamps from shared memory, avoiding costly context switches. However, Parallaft currently cannot record or replay interactions with shared memory. As a result, for both rseq and vDSO, Parallaft masks them, allowing the program to fall back to alternative code paths.

## 4.4 Program-State Checking

At the end of each segment, Parallaft compares the state between the checkpoints taken from the main process and the checker. A mismatch indicates an error within the execution of the main or the checker. In response, Parallaft terminates the application and reports the mismatch.

The program state includes all registers and all addressable memory. Instead of comparing all addressable memory, Parallaft tracks and compares only modified memory pages, since data in unmodified pages must be the same, as they share the same frame in the physical memory.

***Dirty-Page Tracking.*** On x86_64, Parallaft tracks modified pages using the soft-dirty mechanism [1]. On AArch64, Parallaft tracks them by counting the number of maps for each page using a modified kernel implementation of the `PAGEMAP_SCAN` ioctl handler. If a page is mapped exactly once, it is not shared with other processes, hence it is modified or new and should be included for comparison. Otherwise, if a page is mapped more than once, it shares the same underlying frame among the main, the checker and the checkpoint processes, indicating the page is not modified.

***Memory Comparison.*** Once Parallaft gathers the modified memory ranges, it compares the contents of these memory regions between the end-segment checkpoint and the checker. Due to the limitations of being in user-space, Parallaft cannot directly read or map other processes' memory without copying. To avoid this, Parallaft injects hash-computing code into the target processes, hashing all modified memory. Parallaft then compares the hashes only. We use xxHash [6] (XXH3-64b variant) for performance.

## 4.5 Checker Execution Scheduling and Pacing

On a heterogeneous processor, big and little cores can have significantly different performance characteristics for different workloads. For example, when the L1d cache of little cores is much smaller than that of big cores, a non-memory-intensive workload may only see a 2× slowdown on a little core, whereas a memory-intensive workload may experience a 8× slowdown. Suppose we only have four little cores. On the one hand, checking the non-memory-intensive workload will, on average, result in two cores idling while the other two cores run at full speed, which is less power efficient than running all four cores at a lower DVFS point. On the other hand, checking the memory-intensive workload will fail to keep up with the speed of the main core, hence slowing down the overall execution.

For power efficiency, Parallaft dynamically scales the little core's DVFS point so that their total processing power is just enough to cover the need by the checkers[10].

Conversely, if the checkers run out of little cores, to avoid throttling the main execution, Parallaft automatically migrates the oldest checker to one of the big cores, hence freeing up one little core, so that the newest checker can run on it (figure 4).[11] Since this means the program checkers can then keep up with the application, this mitigates slowdown from checkpoints building up and still being live at the end of the main process' execution. Similarly, any checkers that

[8]On x86_64, Parallaft's prototype does not yet intercept other nondeterministic instructions, such as `rdrand`, `rdpid`, and SGX instructions, due to the lack of hardware capability to generate faults on these instructions and the difficulty in patching instructions in a variable-length instruction set. It is however simple to patch `cpuid`'s output to a version of x86_64 where these instructions appear to be unavailable to the application if needed.

[9]Reading system registers on and above EL1, such as `MIDR_EL1`, will result in a trap to kernel, but the kernel does not have facilities to deliver a signal to the process on these traps. Kernel support would marginally simplify Parallaft by avoiding the need for binary patching.

[10]Standard governors, while able to choose DVFS points, do a poor job of scheduling Parallaft's checkers because they are not aware that despite being compute-bound, our checkers rarely have short-term latency requirements, so they end up running at maximum clock speed unnecessarily.

[11]We considered instead scheduling the newest checker to a big core instead. However, this was unnecessary for performance, as just one big core can keep up, and only added to memory pressure by increasing the number of live segments and reduced energy efficiency by using the big core for a full checkpoint instead of only part of one.

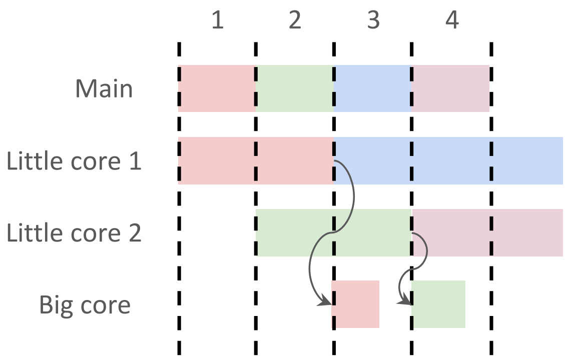


**Figure 4.** When Parallaft exhausts all little cores, the oldest checker is migrated to a big core, running in an energy-inefficient way briefly in order to keep up with the main execution rather than queuing work for later.

**Table 3.** Experimental setup.

| *Hardware* | |
|---|---|
| Machine | Apple Mac Mini |
| CPU | Apple M2 (AArch64) with 4 little cores (Blizzard-M2) & 4 big cores (Avalanche-M2) |
| Memory | 16 GB LPDDR5 with 16 GB swap |
| *Software* | |
| OS | Ubuntu Asahi Linux 24.04 |
| Kernel | 6.10.5-asahi |
| Compiler | GCC 13.2.0 with `-O3` optimization |

are still running at the end of the main execution (which cannot be determined in advance) increase whole-program execution time, so are migrated to big cores to finish quickly.

## 5 Evaluation

Parallaft only incurs 15.9% performance overhead and 44.3% energy overhead, compared with 16.2% performance overhead and 87.8% energy for RAFT under like-for-like threat models (section 3.4) on our AArch64 machine.

### 5.1 Experimental Setup

To evaluate the performance, energy, and memory overhead, we run SPEC CPU2006 with *ref* datasets (running all inputs for those with multiple) on the AArch64 system in table 3. Each benchmark is run three times unless otherwise noted, and all run successfully. In addition, we evaluate on a second platform, an Intel x86_64 heterogeneous processor.

To measure energy overhead, we read power consumption of the SoC and the DRAM (excluding the GPU) from the system management controller (SMC) at an interval of one second during benchmark execution. We then integrate each second's energy consumption over the duration of the

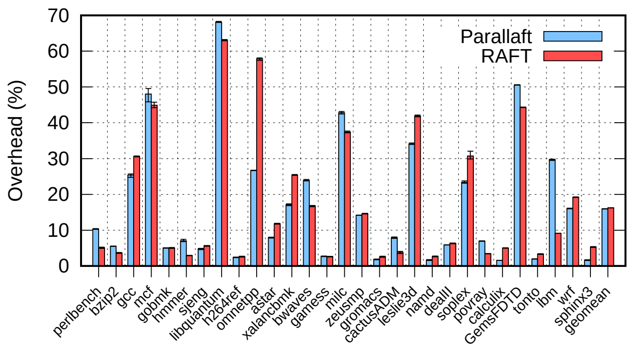


**Figure 5.** Performance overhead of Parallaft and RAFT. Compared with RAFT which has a 16.2% overhead, Parallaft only incurs an overhead of 15.9%.

benchmark to get total energy consumed. To measure memory overhead, every half second during benchmark execution, we sample the sum of the proportional set size (PSS)[12] of main, checker, and runtime processes. Each benchmark is run only once for memory measurements to save time.

Since RAFT [55] has no public release, we modify Parallaft as follows to model RAFT's overheads. (1) *No periodic checkpoints.* We disable automatic program slicing but instead only take a checkpoint at the very beginning and the very end of the program, so we have a single segment throughout the program execution (except during handling of file-backed `mmap` syscalls, two checkpoints are taken, in order to duplicate the file descriptor to the checker process). (2) *Homogeneous execution.* We run main and checker processes on big cores, so that the checker will not systematically run slower than the main process. (3) *No state comparison.* We disable state comparison and dirty page tracking at the end of each segment, which are not present in RAFT.

### 5.2 Performance Overhead

As shown in figure 5, Parallaft incurs geometric mean performance overhead of 15.9%. This compares favorably with RAFT, which has performance overhead of 16.2%.

The performance overhead of RAFT is mainly due to resource contention between the main and the checker processes, as both run on big cores with a shared L2 cache. The performance overhead is especially profound in memory-intensive benchmarks, such as `mcf` and `milc`, where L2 cache contention is more severe.

Parallaft suffers less resource contention overhead as the checkers run on little cores, which share a separate L2 cache from big cores, though it still suffers from DRAM contention like RAFT. However, Parallaft introduces other performance overheads, such as the cost of forking and copy-on-write

[12]PSS is the amount of memory shared with other processes divided by the number of processes sharing that memory. Unlike resident set size (RSS), summing multiple processes' PSS provides a more accurate measure of total memory usage when significant memory is shared via copy-on-write.

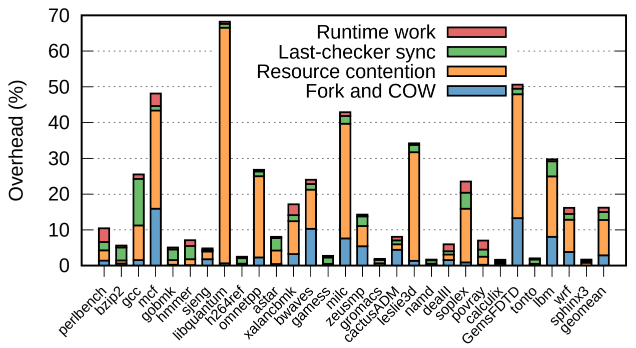


**Figure 6.** Performance-overhead breakdown of Parallaft. For most benchmarks, resource contention and fork-and-copy-on-write dominate performance overhead.

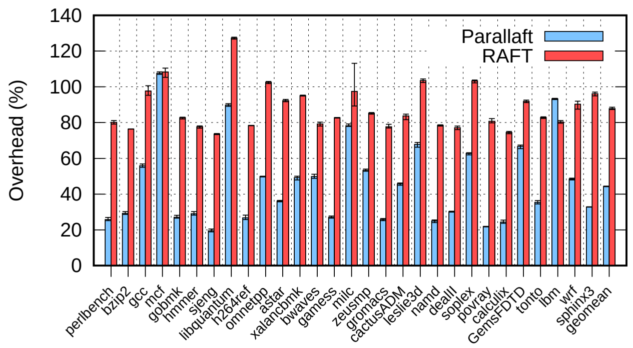


**Figure 7.** Energy overhead of Parallaft and RAFT. Parallaft incurs half energy overhead compared with RAFT.

(COW), dirty page tracking, and waiting for the last checkers to finish after the main finishes.

**5.2.1 Overhead Breakdown.** Figure 6 breaks down the performance overhead of Parallaft into the following areas. (1) *Forking and COW.* The cost of forking the main process at the beginning of each segment, and the COW operations if the main process first writes to a page after the beginning of a segment. These operations cannot be removed from the critical path, and hence contribute to the performance overhead. This is measured by the difference of the *system* CPU time in the baseline run and the Parallaft run. (2) *Resource contention.* The cost of resource contention between the main and the checker execution, due to shared last-level cache (LLC) and shared memory bandwidth. Checker migration to big cores also contends L2 cache shared by them. Resource content overhead is measured by the difference of the *user* CPU time in the baseline run and the Parallaft run. (3) *Last-checker sync.* Cost of waiting for the last checkers to finish after the main process finishes, measured by the difference of the execution time of the main process and the execution time of all checker processes. (4) *Runtime work.* Work that cannot be removed from the critical path, such as clearing dirty bits at the start of each segment for dirty-page tracking, setting up performance counters to enable execution point recording, capturing the data from the syscalls made by the main program for syscall comparison and replay, calculated by the difference of total overhead and the sum of the above three components.

The dominating overhead is resource contention for most benchmarks, due to limited capacity in the shared LLC and limited shared memory bandwidth. Benchmarks with intensive memory use have higher resource contention overhead, such as `mcf`, `milc` (as previously mentioned for RAFT) and `libquantum`. In addition, resource contention is even more severe if checkers cannot keep up with the main and hence constantly migrate to big cores, as migration to big cores result in pollution of their cache, which is the case in `mcf`, `milc` and `lbm`. In these benchmarks, checkers do 41.7%, 38.0%, and 50.0% of work on big cores, respectively. The second highest overhead is forking and COW, which also depends on the memory intensity of the workload. The forking and COW overhead will be high if the workload modifies large numbers of pages in each segment.

The other two categories, last-checker sync and runtime work, are small in most benchmarks. However, last-checker-sync overhead is more significant for benchmark runs split into multiple short processes, such as `bzip2`, `gcc` and `soplex`.

## 5.3 Energy Overhead

Figure 7 shows the energy overhead of Parallaft and RAFT. Parallaft has an average energy overhead of 44.3%, about half of the energy overhead of RAFT (87.8%), thanks to the little cores' energy efficiency. In contrast, as RAFT checkers run on big cores, they need nearly the same amount of energy required to run the main program, making RAFT's normalized energy consumption nearly 2×.

The energy benefit of Parallaft is less significant in benchmarks where the checkers on little cores constantly fall behind the main execution, hence requiring migration to big cores, such as `mcf`, `milc`, and `lbm`. `lbm` is the only benchmark in which Parallaft has higher energy overhead than RAFT, where checkers do half their work on big cores to keep up.

The additional work of forking and COW operations in Parallaft also adds to the energy cost. In most benchmarks, the gain from running checkers on little cores is still significant enough to outweigh the extra energy cost.

## 5.4 Memory Overhead

Figure 8 shows normalized memory consumption of Parallaft, geomean 3.32×. This is the memory consumption of main, checker, and runtime processes normalized against the baseline consumption, averaged over time. We exclude private memory used by checkpoints as they can be swapped out to disk without affecting the performance of the program in most cases, because the memory in each checkpoint

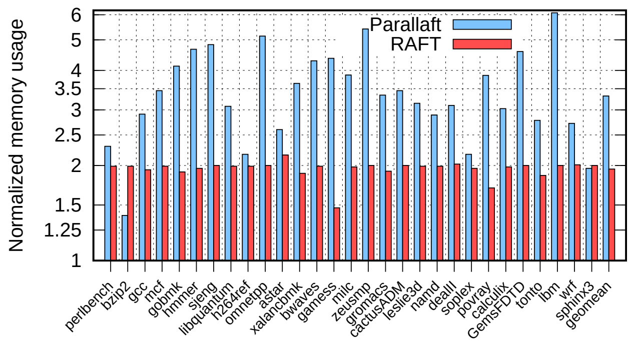


**Figure 8.** Normalized memory usage of Parallaft and RAFT.

will only be accessed once during program-state comparison against the checker, which is not on the critical path.

On average, Parallaft uses more memory than RAFT (3.32× vs 1.95×). This is unsurprising, as it is mainly caused by deliberately maintaining more copies of the program execution than RAFT to exploit heterogeneous parallelism [8].

## 5.5 Parameter Sensitivity

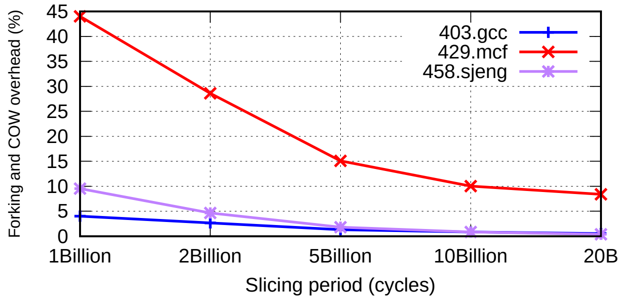
Our slicing-period parameter can be tuned to trade off forking and copy-on-write overheads against last-checker-sync overhead. Here we study parameter sensitivity.

We can reduce forking-and-COW overhead with a longer slicing period, as the main process will trigger fewer fork and COW operations in total. However, the last-checker-sync overhead will increase, due to the increased lag between the main and the checker execution.

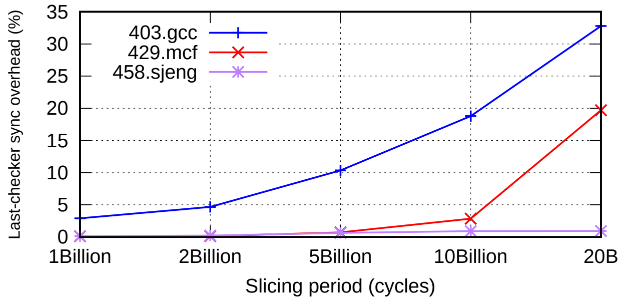
To evaluate the effect of slicing period on performance overhead, we select three benchmarks: `gcc`, `mcf` and `sjeng`. `gcc` has a short execution time in each part of the benchmark, hence the last-checker-sync overhead is more significant. `mcf` is memory-intensive so modifies a large number of pages each segment, resulting in high forking-and-COW overhead. In contrast, `sjeng` is a benchmark with moderate characteristics. We run them under Parallaft with slicing periods of 1 billion to 20 billion cycles, and measure the performance overhead, including last-checker-sync and forking-and-COW components. Results are shown in figure 9.

As shown in figure 9(a), a lower slicing period leads to higher forking-and-COW overhead, with the effect being more significant in memory-intensive benchmark `mcf`, as more pages are duplicated due to COW.

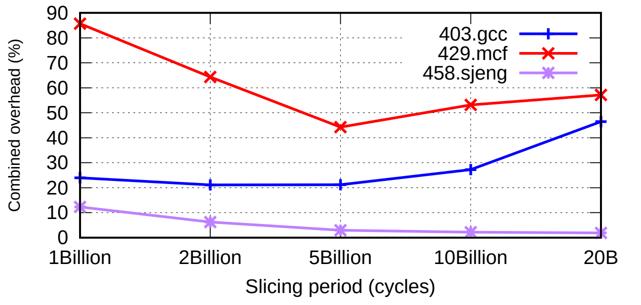
Moreover, as shown in figure 9(b), a longer slicing period results in higher last-checker-sync overhead, due to the increase of the lag between the main and the checker execution. The proportion of last-checker-sync overhead is more profound in benchmarks with shorter execution times, such as `gcc`, where the 9 inputs run in under 87 seconds in the baseline, which translates to less than 10 seconds for each main execution (input). Moreover, last-checker-sync overhead is more sensitive to slicing period in benchmarks when



**(a)** Forking-and-COW overhead vs. slicing period. Forking and COW overhead decreases as the slicing period increases due to fewer memory pages being COW-ed. The decrease is more significant in memory-intensive benchmarks such as `mcf`.



**(b)** Last-checker-sync overhead vs. slicing period. Last-checker-sync overhead increases with slicing period from increased lag between main and checker execution. This is prominent in benchmarks with short execution time (`gcc`), or where each checker on a little core has a large slowdown compared with the big core (`mcf`).



**(c)** Combined performance overhead vs. slicing period. Due to the counteracting effect of forking-and-COW overhead and last-checker-sync overhead, there is a sweet spot where the combined performance overhead is the lowest for each benchmark.

**Figure 9.** Slicing-period performance tradeoffs.

little cores are significantly slower than the big core, such as `mcf`, where a little core has a slowdown of more than 4×. In contrast, the last-checker-sync overhead of `sjeng` is very insensitive to the slicing period, because it takes the longest to run (226 s in the baseline) among the three benchmarks. In addition, in `sjeng`, each little core only has a 2.0× slowdown, hence leaving less work to do after main finishes.

Due to the combination of effect of slicing period on forking and COW and last checker sync overhead, there is a

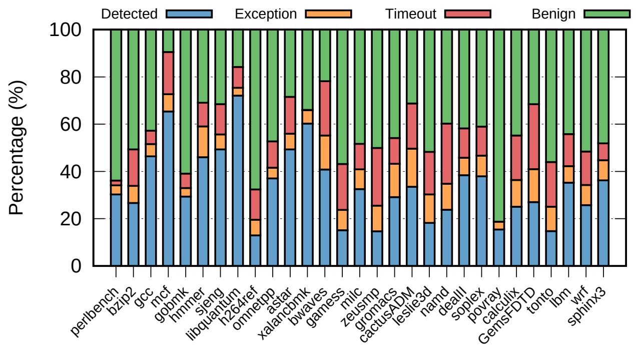


**Figure 10.** Error-injection results, split between errors that do not affect correctness and are ignored (benign), and three detection scenarios (detected, exception, timeout).

sweet spot for each benchmark, where the performance overhead is minimized. For `gcc`, `mcf`, and `sjeng`, the sweet spot is at 2 billion, 5 billion and 20 billion cycles respectively.

### 5.6 Fault Injection

To verify Parallaft's ability to detect faults, we inject faults into the checkers and record the response of Parallaft. We first perform a profile run of the checker in each segment to get an execution time $t$ of the checker, without injection faults. After that, for each segment, the checker is run another five times. Each time, a fault is injected at a random point in the execution: by sleeping $t'$, which is selected uniformly from $[0, 1.1t)$, before the injection. As injections may fail because the checker may finish earlier than expected, we discard failed injections and try again. The fault we inject is a random bit flip in a random register, selected from the general-purpose, floating-point and vector registers.

We classify the outcome of each fault injection as follows. (1) *Detected.* The fault is detected by Parallaft, excluding exceptions, which are separately accounted. (2) *Exception.* The fault causes an exception in the checker, which is a special case of *Detected.* (3) *Timeout.* The checker has executed more instructions than possible in the main execution of the segment, another *Detected* class (section 4.2.2). (4) *Benign.* The fault does not cause observable effects and the program can still finish with correct output.

We perform the fault injection experiment on all benchmarks in SPEC CPU2006, with results shown in figure 10. On average, 43.3% errors are benign, meaning they neither affect the program output nor diverge the state at segment-end checks, while all other errors are detected by Parallaft. Because it duplicates all computation, Parallaft gives 100% fault coverage for user-space execution of applications for single-event upsets (SEUs), bar hash collision[13] or repeated errors. While Parallaft's runtime and the operating system are not protected (section 3.4), the runtime only uses 0.26% of CPU time compared relative to the main and the checker execution, hence the chance of missing a fault is low.

### 5.7 Syscall and Signal Handling Overhead

Parallaft incurs overhead when handling syscalls and signals. We evaluate in the extreme scenarios where syscalls and signals are stress-tested.

For syscalls, when `getpid` is repeatedly called, Parallaft introduces a slowdown of 124.5×, which is dominated by `ptrace` operations. When blocks of 1 MB of data are repeatedly read from `/dev/zero`, the slowdown is 18.5×, which is dominated by recording the data read. RAFT incurs almost identical slowdown because of shared syscall-handling logic. To reduce syscall overhead, it is future work to develop an in-process syscall-interception technique [37]. For signals, when `SIGUSR1` is raised repeatedly with an empty signal handler, Parallaft brings a slowdown of 39.8×.

### 5.8 Overhead on Intel x86_64

In addition to our Apple M2 processor, we also evaluate on an Intel heterogeneous processor for completeness.

***Experimental Setup.*** We use a Intel Core-i7 14700 machine with 128 GB of DDR5 RAM. We set the slicing period to 5 billion instructions[14]. To measure energy overhead, we read CPU package energy from the Intel Running Average Power Limit (RAPL) interface. The same set of benchmarks is used as in our experiments on the Apple processor.

***Results.*** Parallaft incurs geomean performance overhead of 26.2%, whereas RAFT's is 12.9%. This is because smaller 4 KB page sizes on Intel, versus 16 KB on Apple, make Parallaft's checkpointing more expensive, and because we see more cache contention from the many competing threads than on Apple. Moreover, the little cores on Intel are less energy-efficient than on Apple due to the lack of a separate voltage domain from the big cores. Parallaft incurs geomean energy overhead of 46.7% whereas RAFT's is 50.2%.

## 6 Conclusion

We have presented Parallaft, a runtime-based system for CPU fault tolerance via heterogeneous parallelism. By combining copy-on-write checkpoints, execution point record-and-replay capability, optimized program-state checking, and intelligent scheduling of checkers, Parallaft enables CPU error detection for binary programs at low performance and energy overhead on commodity heterogeneous processors. Additional data related to this paper are available in the repository at https://doi.org/10.5281/zenodo.14084708 [54]. The source code for Parallaft is available at https://github.com/CompArchCam/parallaft.

[13]Parallaft's hash, XXH3-64b, has extremely low collision rates [7], a chance of $3.13 \times 10^{-9}$.

[14]Instead of 5 billion *cycles*, because it can break partially executed rep-prefixed instructions, which are not supported by Parallaft's execution point record and replay feature yet.

## A Artifact Appendix

### A.1 Abstract

Our artifact includes the source code of Parallaft and scripts to reproduce key performance and energy overhead results (Figures 5 to 7) from the paper. To save time, each benchmark is run once instead of three times as in the paper. For accurate reproduction of results, we strongly recommend an Apple M2 processor. If such hardware is not available, our experiments can run on a 12/13/14th-gen Intel x86_64 heterogeneous processor or a Linux-capable Apple Silicon aarch64 processor with some caveats. In addition, at least 16 GB or RAM is required. For x86_64, software requirements include Ubuntu 24.04 with Linux kernel version 6.7 or above. For Apple Silicon aarch64, Asahi Ubuntu 24.04 is required along with a custom kernel. We use SPEC CPU2006 benchmarks (not provided due to licensing) to evaluate performance and energy overhead of Parallaft.

### A.2 Artifact check-list (meta-information)

- **Algorithm**: Dynamically replicating execution of binary programs to detect processor errors.
- **Program**: SPEC CPU2006 benchmark is required but not provided. Download size is approximately 2.6 GB.
- **Compilation**: A script is provided to build Parallaft in a Docker container without manual installation of compilers. GCC/G++/GFortran compilers that support C99, C++98 and Fortran-95 standards are required for SPEC CPU2006.
- **Binary**: Parallaft binary will be built from source during the experiments.
- **Run-time environment**: Parallaft needs x86_64 or aarch64 Linux environment, with kernel version 6.7 or above. On aarch64, a custom 6.10 kernel is needed to read Apple M2 power consumption numbers and to read dirty pages. On x86_64, a stock kernel works. To build Parallaft program, Docker is required. Additionally, to run the experiments, Python 3, curl, and gnuplot are required. Root access on a baremetal machine is needed to enable performance counter access and to dynamically adjust CPU frequency. Virtual machines are not supported.
- **Hardware**: An Apple M2 machine is strongly recommended to reproduce the performance and energy results. Alternatively, any Linux-capable Apple Silicon processors or heterogeneous Intel x86_64 processors also work, but with some caveats (Appendix A.3.1). At least 16 GB of RAM is required. Moreover, we require access to power sensors, performance counters, and CPU frequency control.
- **Run-time state**: Parallaft is sensitive to cache contention.
- **Execution**: At least 24 GB of available RAM plus swap is required. During the execution, main processes are pinned to a big core while checker processes are pinned to little cores. To isolate from noise, we recommend that you are the sole user of the machine during experiment execution.
- **Metrics**: Execution time (both x86_64 and aarch64) and energy consumption (aarch64 only) will be evaluated.
- **Output**: Data is generated for Figures 5 and 6 (x86_64 and aarch64) and Figure 7 (aarch64 only). Each benchmark is executed once, instead of three times as in the original paper, to save time.
- **Experiments**: Scripts are provided to run the experiments.
- **How much disk space required (approximately)?**: 12 GB.
- **How much time is needed to prepare workflow (approximately)?**: 1-2 hours.
- **How much time is needed to complete experiments (approximately)?**: 6-7 hours.
- **Publicly available?**: Yes.
- **Code licenses (if publicly available)?**: BSD license.
- **Archived (provide DOI)?**: 10.5281/zenodo.14084708

### A.3 Description

#### A.3.1 How delivered.

The artefact containing the source code of Parallaft and scripts to run the experiments is accessible on the following link: https://github.com/CompArchCam/reproduce-parallaft-paper.

#### A.3.2 Hardware dependencies.

An Apple M2 processor with at least 16 GB of RAM is strongly recommended to reproduce the performance and energy results in the paper. We require access to the power management IC to read power consumption, branch-counting performance counters for execution point record and replay, as well as the CPU frequency control to maximize energy saving. Alternatively, any other Linux-capable Apple Silicon processors with at least 16 GB of RAM also works, but we do not provide a mechanism to read power consumption numbers. Alternatively, on x86_64, 12/13/14th-gen heterogeneous Intel processors (e.g. Core i7-12700) may be used for performance overhead evaluation. However, as it does not have separate voltage domains for big and little cores, the energy overhead evaluation will not show the expected benefit of Parallaft. Moreover, running Parallaft on Intel incurs more performance overhead due to its smaller page sizes (4 KB on Intel versus 16 KB on Apple, hence introducing more checkpointing overhead), and due to more severe cache contention from the many competing threads.

#### A.3.3 Software dependencies.

***OS distribution.*** Ubuntu 24.04 (on x86_64) or Ubuntu Asahi 24.04 (on Apple Silicon aarch64) is required to run the experiments. The standalone Parallaft (without experimental setup/execution/results collection/plotting scripts) can run on any Linux distribution.

***Kernel.*** On Apple Silicon aarch64, a custom Linux 6.10 kernel is required for Parallaft to read Apple M2 power consumption numbers and dirty pages. A script is provided in the artefact to download, build, and install such kernel. On x86_64, a stock kernel with version 6.7 or above is needed.

***Software packages.*** Docker is required to build Parallaft binary. gnuplot and Python 3 with `subprocess_tee`, `dataclasses_json`, `filelock`, `prctl`, `numpy` packages are required to execute the experiments and collect and plot the results.

***Benchmark.*** SPEC CPU2006 version 1.2 is required (but not included in the artefact package). During its installation process, `curl` is required to download src.alt files and aarch64 SPEC tools. To build the benchmarks, GCC/G++/GFortran compilers that support C99, C++98 and Fortran-95 standards are required.

## A.4 Installation

Ensure that you are running Ubuntu 24.04 (on x86_64) or Asahi Ubuntu 24.04 (on Apple Silicon aarch64).

***Setting up the artefact package.*** Clone the artefact package and install software dependencies with the following commands:

```
$ git clone
    https://github.com/CompArchCam/reproduce-parallaft-
    paper --recursive
$ cd reproduce-parallaft-paper/scripts
$ ./deps.sh
```

Log out and log back in enable Docker access without sudo. Then build the Parallaft binary with the following commands:

```
$ cd reproduce-parallaft-paper/scripts
$ ./build_app.sh
```

Get a copy of SPEC CPU2006 version 1.2. Place the ISO file, `cpu2006-1.2.iso`, at the root of the artefact package (where a `PLACE_SPEC_ISO_HERE` file is co-located). Then install it with the following commands:

```
$ ./install_spec06.sh
```

***Installing the custom kernel for Apple Silicon aarch64.*** On Apple Silicon aarch64 only, download, build, and install the custom kernel and reboot the machine for the new kernel to take effect with the following commands:

```
$ ./build_kernel.sh
$ sudo reboot
```

After the reboot, check if you are running the correct kernel:

```
$ cd reproduce-parallaft-paper/scripts
$ ./check_kernel.sh
```

You should see it reporting the kernel version and the power sensor being OK.

## A.5 Experiment workflow

In our experiments, the following runs are performed on all SPEC CPU2006 int and fp benchmarks.

- A baseline run, to get execution time and CPU time without Parallaft or RAFT.
- A baseline energy-consumption profiling run, to get baseline energy consumption without Parallaft or RAFT.
- A Parallaft run.
- A RAFT run.

To run all the experiments above and plot the results, run `./run.sh` and `./plot.sh` in the `scripts` directory of the artefact package.

## A.6 Evaluation and expected result

Our experiments reproduce the following results under `plots` directory. Since each benchmark is only run once, no errors bar will be shown as in the original paper.

- Performance overhead of Parallaft and RAFT (Figure 5).
- Performance-overhead breakdown of Parallaft (Figure 6).
- Energy overhead of Parallaft and RAFT (Figure 7). This result will only be available on Apple Silicon aarch64 platforms.

Since Parallaft runs redundant copies of the program on additional CPU cores, due to cache contention, the performance overhead can vastly differ on processors with different cache organizations. On our Apple M2 Mac Mini, Parallaft incurs performance overhead of 15.9% and energy overhead of 44.3%.

## A.7 Experiment customization

***Running a subset of benchmarks or experiments.*** Tweak `BENCHMARKS` and `EXPERIMENTS` in `scripts/run.sh`.

***Tuning parameters.*** To change checkpoint period (in number of CPU cycles), tweak `PARALLAFT_CHECKPOINT_PERIOD` in `scripts/run.sh`.

***Running an arbitrary program under Parallaft.*** To run an arbitrary program under Parallaft on an Apple M2 processor, run the following command under the artefact package:

```
$ ./bin/parallaft --config
    ./app/parallaft/configs/apple_m2_fixed_interval.yml --
    path/to/your/program arg1 arg2
```

When the execution finishes, Parallaft dumps some statistics. The key ones are:

- **`timing.all_wall_time`**: Wall time elapsed to finish the program execution, including the waiting time for outstanding checkers to finish after the main finishes.
- **`timing.main_wall_time`**: Wall time elapsed to finish the main program execution, not including the waiting time for checkers.
- **`timing.main_{user,sys}_time`**: User/system time used by the main program execution.
- **`hwmon.macsmc_hwmon/*`**: (Apple Silicon only) Energy used for different components of the SoC during the program execution.
- **`counter.checkpoint_count`**: Number of checkpoints taken, including checkpoints taken to handle certain `mmap` syscalls and to slice the program execution for checker parallelism.
- **`fixed_interval_slicer.nr_slices`**: Number of segments sliced due to reaching the specified checkpoint period.

Use another config to run on a Intel processor, e.g. for Intel Core i7-12700, use `intel_12700_fixed_interval.yml`.

***Running Parallaft on a different processor.*** Parallaft itself relies on accurate branch-counting hardware performance counter for the execution point record and replay capability. In addition, in our experiments, the big/little core configuration of the processor needs to be known. Moreover, a way to read processor power consumption is required. Check `docs/hardware_support.md` file in the artefact package for instructions to run Parallaft on a different processor.

## A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

## References

[1] 2013. Soft-dirty PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt

[2] 2015. Restartable sequences. https://lwn.net/Articles/650333/

[3] 2017. AMD EPYC Brings New RAS Capability. https://www.amd.com/system/files/2017-06/AMD-EPYC-Brings-New-RAS-Capability.pdf

[4] 2023. perf_event_open(2) — Linux manual page. https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[5] 2024. ptrace(2) - Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html

[6] 2024. xxHash. https://xxhash.com/

[7] 2024. xxHash: Collision ratio comparison. https://github.com/Cyan4973/xxHash/wiki/Collision-ratio-comparison#collision-study

[8] Sam Ainsworth and Timothy M. Jones. 2018. Parallel Error Detection Using Heterogeneous Cores. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2018.00044

[9] Sam Ainsworth and Timothy M. Jones. 2019. ParaMedic: Heterogeneous Parallel Error Correction. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2019.00032

[10] Sam Ainsworth, Lionel Zoubritzky, Alan Mycroft, and Timothy M. Jones. 2021. ParaDox: Eliminating Voltage Margins via Heterogeneous Fault Tolerance. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA51647.2021.00051

[11] Arm Ltd. 2023. *Arm Cortex-X2 Core Technical Reference Manual. Cache protection behavior.* https://developer.arm.com/documentation/101803/0200/RAS-Extension-support-/Cache-protection-behavior

[12] T.M. Austin. 1999. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.1999.809458

[13] David F. Bacon. 2022. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. In *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*.

[14] S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* (2005). https://doi.org/10.1109/MM.2005.110

[15] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Computing Surveys (CSUR)* (2015). https://doi.org/10.1145/2790077

[16] Matthew Connatser. 2024. Game dev accuses Intel of selling 'defective' Raptor Lake CPUs. https://www.theregister.com/2024/07/13/game_raptor_intel/.

[17] Jonathan Corbet. 2011. On vsyscalls and the vDSO. https://lwn.net/Articles/446528/

[18] Irving Baysah Daniel Henderson. 2021. *Introduction to IBM Power® Reliability, Availability, and Serviceability for POWER9® processor-based systems using IBM PowerVM™ with updates covering the latest Power10 processor-based systems.* Technical Report. IBM Systems Group. https://www.ibm.com/downloads/cas/2RJYYJML

[19] Moslem Didehban, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. 2017. InCheck: An In-Application Recovery Scheme for Soft Errors. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/3061639.3062265

[20] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A compiler technique for near Zero Silent Data Corruption. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1145/2897937.2898054

[21] Moslem Didehban and Aviral Shrivastava. 2018. A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments. *IEEE Transactions on Reliability* (2018). https://doi.org/10.1109/TR.2018.2793098

[22] Moslem Didehban, Hwisoo So, Prudhvi Gali, Aviral Shrivastava, and Kyoungwoo Lee. 2024. Generic Soft Error Data and Control Flow Error Detection by Instruction Duplication. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2024). https://doi.org/10.1109/TDSC.2023.3245842

[23] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. arXiv:2203.08989 [cs.AR]

[24] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. arXiv:2102.11245 [cs.AR]

[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*. https://doi.org/10.1145/844128.844148

[26] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores That Don't Count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. https://doi.org/10.1145/3458336.3465297

[27] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4

[28] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. 2019. The Arm Triple Core Lock-Step (TCLS) Processor. *ACM Transactions on Computer Systems (TOCS)* (2019). https://doi.org/10.1145/3323917

[29] Casey M. Jeffery and Renato J. O. Figueiredo. 2012. A Flexible Approach to Improving System Reliability with Virtual Lockstep. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2012). https://doi.org/10.1109/TDSC.2010.53

[30] Xu Liu and John Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2013.6557169

[31] Albert Meixner, Michael E. Bauer, and Daniel Sorin. 2007. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2007.18

[32] J. M. Mellor-Crummey and T. J. LeBlanc. 1989. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/70082.68189

[33] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M. Jones. 2016. COMET: Communication-optimised multi-threaded error-detection technique. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. https://doi.org/10.1145/2968455.2968508

[34] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2007.35

[35] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2002.1003566

[36] Khang T Nguyen. 2017. New Reliability, Availability, and Serviceability (RAS) Features in the Intel Xeon Processor Family. https://www.intel.com/content/www/us/en/developer/articles/technical/new-reliability-availability-and-serviceability-ras-features-in-the-intel-xeon-processor.html.

[37] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*.

[38] A. Robert Pargeter. 1996. An example of strong induction. *The Mathematical Gazette* 80, 488 (1996), 406–407. https://doi.org/10.2307/3619594

[39] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/1772954.1772958

[40] S.K. Reinhardt and S.S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/342001.339652

[41] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2005.34

[42] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2002.1028924

[43] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2007.98

[44] T.J. Slegel, R.M. Averill, M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. 1999. IBM's S/390 G5 microprocessor design. *IEEE Micro* 19, 2 (March 1999), 12–23. https://doi.org/10.1109/40.755464

[45] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.23919/DATE.2018.8342065

[46] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. 2004. The impact of technology scaling on lifetime reliability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2004.1311888

[47] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream processors: improving both performance and fault tolerance. *SIGARCH Computer Architecture News* 28, 5 (Nov. 2000), 257–268. https://doi.org/10.1145/378995.379247

[48] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1950365.1950370

[49] T.N. Vijaykumar, I. Pomeranz, and K. Cheng. 2002. Transient-fault recovery using simultaneous multithreading. In *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2002.1003565

[50] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. 2010. ParaLog: enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1736020.1736051

[51] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. 2013. Nondeterminism and overcount on modern hardware performance counter implementations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2013.6557172

[52] Neil Werdmuller. 2021. *Addressing functional safety applications with ARM Cortex-R5*. https://community.arm.com/groups/embedded/blog/2015/01/22/addressing-functional-safety-applications-with-arm-cortex-r5

[53] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2451116.2451153

[54] Boyue Zhang, Sam Ainsworth, Lev Mukhanov, and Timothy Jones. 2024. *Artefact evaluation package for "Parallaft: Runtime-based CPU Fault Tolerance via Heterogeneous Parallelism"*. https://doi.org/10.5281/zenodo.14172159

[55] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Runtime Asynchronous Fault Tolerance via Speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2259016.2259035

[56] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. 2010. DAFT: Decoupled acyclic fault tolerance. In *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/1854273.1854289

[57] Y. Zu, C. R. Lefurgy, J. Leng, M. Halpern, M. S. Floyd, and V. J. Reddi. 2015. Adaptive guardband scheduling to improve system-level efficiency of the POWER7+. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830824

Received 2024-09-12; accepted 2024-11-04