ParaVerser: Harnessing Heterogeneous Parallelism for Affordable Fault Detection in Data Centers

Minli Julie Liao
University of Cambridge
ml2076@cam.ac.uk

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

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

Adrian Barredo
Barcelona Supercomputing Center
adrian.barredof@gmail.com

Markos Kynigos
University of Manchester
m.kynig@gmail.com

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

***Abstract*—Data-center operators have awoken to the fact that silent data corruption resulting from defective silicon compute units is endemic at scale. Software scanners have been deployed to mitigate the issue, but either have low coverage or take months, leaving long windows of incorrect behaviour. By contrast, the redundancy mechanisms used in automotive double the required power and area, so cannot be practically deployed in server-space.**

We present ParaVerser, a high-coverage, low-overhead solution to hardware-level error detection in servers. Through minor architectural modifications, we enable conventional cores in heterogeneous server-grade processors to act as checker cores, thus exploiting heterogeneity, frequency scaling and the inherent parallelism in repeat runs to provide energy-efficient error checking. By dynamically coupling big.LITTLE-style out-of-order superscalar cores with in-order ones, we reduce energy overheads relative to a typical lockstep system by 70% with identical guarantees, at only 4.3% performance degradation, and 1064B per-core area overhead.

I. Introduction

The threat of silent data corruption at data-center scale is becoming too frequent to ignore. Both Meta [1] and Google [2] have sounded the alarm that, as transistors get smaller and distributed computations bigger, CPUs persistently produce incorrect results with no warning, crashes or outward signs of error, in spite of existing Reliability, Availability and Serviceability (RAS) mechanisms [3]–[5].

The solution space to tackle reliability has diverged, between software diagnostic techniques on one extreme [6], to full hardware lockstep on the other [7]. In the data-center, software scanners [6], [8], [9] to test in-production systems for permanent faults are now used at scale. While these are simple to deploy, they involve removing access to resources for long periods, or only catch a small fraction of faults. Thus it can take up to six months [6] to remove faulty hardware from production. Automotive systems have long deployed full hardware error detection using dual- and triple-core lockstep [7], [10], using cores duplicated and kept perfectly in sync to compare outputs. However, the halving in compute performance for a given area and power budget makes such a strategy unrealistic for data centers.

Still, promising alternatives exist. Repeat runs of computation, for fault tolerance, are strictly more parallel than the original run [11]–[13]. It is possible to split up programs into segments based on the data dependencies in an initial run, and then reconstruct a full fault-tolerant trace of the computation using induction. Each segment can be run on slow, efficient, parallel hardware instead [11]. Heterogeneity between core sizes has started to emerge in server environments [14], [15]. However, proposed error-detection systems use the slowest cores feasible [13], meaning 16 checker cores for every original "main core" that actually runs computation. While this is efficient, tiny checker cores are infeasible for running real applications if the server is being used for workloads that do not need fault tolerance, and real heterogeneous systems have much larger, superscalar "small" cores [14].

We show that error detection can be enabled in server-grade SoCs at the microarchitectural level with adjustable reliability and performance capabilities, resulting in negligible performance overhead when necessary, and overheads minimised via heterogeneous parallelism. We present ParaVerser, a hardware mechanism for *opportunistic* parallel error detection, using spare CPU resources (heterogeneous or homogeneous) to efficiently duplicate computation, with minimal core modification. Our contributions are as follows:

- We present a new microarchitecture design to detect hard and soft hardware faults, reusing the cores available in conventional server-grade chips. Our design adds an induction-parallelism mechanism [11] to split a run into segments, and replay and check the executed code on multiple slower, parallel cores. Each core in a heterogeneous SoC can be used for running workloads or redundant validation.
- We implement two operational modes: i) a full-coverage mode able to capture all permanent and transient errors, and ii) an opportunistic mode, which gives partial coverage by only checking as much computation as resources are free.
- We optimise the design by (i) designing a hybrid Load-Store-Log and data cache to reduce SRAM overheads; (ii) implementing a Load-Store Push Unit to actively push data direct to checker cores, eliminating local cache pressure and coherence overheads from main-core buffering, (iii) enabling speculative out-of-order checking, and (iv) minimizing cross-chip network traffic via new hashing mechanisms.

- We demonstrate that the performance overhead of ParaVerser in full-coverage mode varies from 1% to 4% geomean, depending on type, number and frequency of checker cores, reducing energy overhead from 95% to just 29% between these two extremes. Opportunistic mode introduces only 1% overhead, covering between 94% and 99% of executed instructions depending on resources allocated.
- We evaluate our design and compare against prior works [11], [13] using more detailed and reasonable core models. We demonstrate that 12 dedicated checker cores [11] are insufficient (9% overhead). While 16 dedicated cores [13] achieves low slowdown, it is at the expense of 35% area overhead.

ParaVerser can provide comprehensive detection of hard and soft faults at the microarchitecture level, with minimal impact on performance and area, and at only a third of the energy overhead of homogeneous lockstep. Our design can facilitate hardware predictive maintenance [16] by identifying CPUs that may become error-prone, possibly due to aging [17], before they fail. Its flexibility also allows for automatic disabling of the fault detecting mechanism during periods of high system load. ParaVerser is an ideal solution for data centers seeking the highest level of service quality while also benefiting from reliable fault-detection capabilities.

II. Related Work

Ripple and FleetScanner (section III-A) have been deployed at scale within commercial servers [6]. However, these periodic test mechanisms detect erroneous hardware within the timescale of months and without guaranteed 100% coverage. Bacon [8] describes the challenges and software approaches of detecting SDCs in the Google Spanner database. Serebryany et al. develop Silifuzz [9], which fuzzes software proxies of CPUs (e.g., simulators or disassemblers) to generate representative tests. Harpocrates [18] extends Silifuzz with a hardware-in-the-loop generation method using gem5 simulation, to target specific CPU structures. All four of Ripple, FleetScanner, Silifuzz and Harpocrates can be categorised as Built-In-Self Test [19]–[21], whereas ParaVerser makes full Redundant Execution feasible due to its inherent parallelism strategy. ParaVerser is thus complementary because while BIST is cheap, it cannot a) cover all faults (including transient faults) like our full-coverage mode can, or b) generate representative tests of the actual application code as it is running, like our opportunistic mode can to cover hard faults. BIST executes predeveloped proxies, which will only have good coverage if workloads do not change and proxy inputs are representative. Redundant execution can also be achieved in software, at high overheads [22]–[24]. Several works for HPC look for SDCs via anomaly detection on unusual data outputs [25]–[29], others via selective instruction duplication on soft errors [30]–[32].

Major CPU vendors implement various RAS (Reliability, Availability and Scalability) mechanisms for detecting errors [3], [4], [33], [34], as well as post-silicon validation [35] and power-on self test [19], [20]. IBM's Power architectures use various mechanisms to cover parts of the CPU cores, caches, DRAM, bus, and I/O system [5], [33], such as stacked latches and replay of instructions upon failure, but this requires a more complex CPU design and has overheads in power, area, and performance. Intel has implemented similar error correction and detection mechanisms for the CPU logic as IBM [3], but these mechanisms miss errors in Google and Meta data centers [1], [2]. HPC systems also experience a large number of CPU errors [36]–[39], which have been studied for their impact on micro-architectures [40]–[42].

Fault resilience techniques conventionally deployed in safety-critical systems involve high-overhead dual- and triple-core lockstep, respectively [10], [43]–[46].

Meixner et al. proposed Argus [47] for finding errors in simple cores, by tracking control flow, data flow, computation and memory through orthogonal processes. Though this comes at only 17% hardware overhead, it requires comprehensive changes to the entire microarchitecture, and unlike ParaVerser, the hardware performing the checking cannot be repurposed for standard computation. Razor [48]–[50] employs shadow latches to protect critical-path flip-flops against some forms of timing error, though with vulnerabilities to metastability [51]. Similar to ParaVerser, mSWAT [52] employs a log of loads to achieve checkpoint replayability across threads. However, it only replays computation on certain anomalies (e.g., fatal traps, hangs and panics), rather than checking continuously; so while it catches many errors [53]–[55], it would miss the SDCs that are plaguing the industry, such as the example highlighted by Meta (an FPU returning 0 for some inputs rather than the real value) [1], and cannot catch all such errors even with supplemental detection techniques [56].

Aggarwal et al. [57], followed by Sorin [58] and Gupta et al. [59], [60] approach fault resilience by compartmentalizing and reusing faulty components. The re-use of pre-existing components to reduce additional complexity has also been studied [46], [61]–[63]). DIVA [64] introduces the use of an in-order superscalar checker at the end of an out-of-order pipeline, to repeat each output instruction individually. To reconstruct each instruction into a full run, ECC is required on intermediate states, requiring significant design invasion and affecting critical paths. Heterogeneous Parallel Error Detection [11]–[13] avoids the need for inter-state or register ECC by using a coarser grained form of thread-level parallelism, instead checking that register checkpoints and memory accesses can be recalculated between segments.

Prior works on redundant multithreading or process-level redundancy [65]–[69] achieve error detection through redundant execution on other core or threads, like ParaVerser. However, ParaVerser's method provides heterogeneous parallelism in the redundant execution, resulting in higher power efficiency and low performance overhead.

III. Motivation

A. Errors at Server-Scale

Silicon-level faults within data centers have become endemic. Reports from Meta [1] and Google [2] indicate that systems now frequently suffer from hard faults, often with no

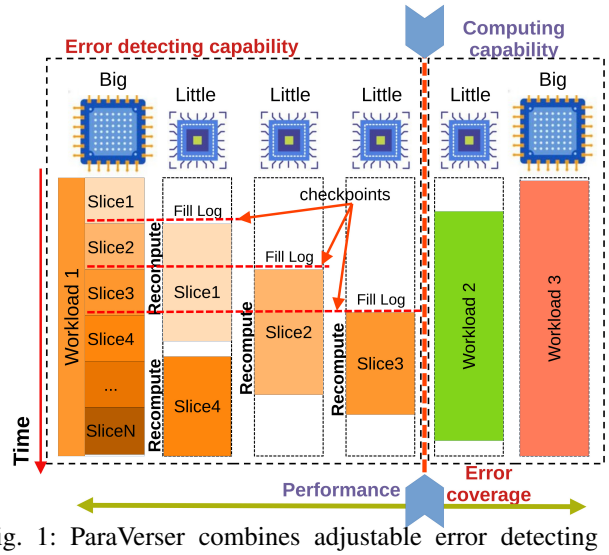

Fig. 1: ParaVerser combines adjustable error detecting and computing capabilities across heterogeneous cores.

crashes or warning signs, resulting in silent data corruption, and escaping the Reliability, Availability and Serviceability (RAS) [3] features built into the processor.

In response, operators have turned to software scanning tools like FleetScanner and Ripple [6]. FleetScanner does *out-of-production testing*, where servers must be placed in maintenance mode, covering the full data center over a 6-month timescale, leaving long windows of vulnerability where errors go undetected but affect real execution. Ripple is *in-production testing*, or smaller tests run at the same time as real workloads, at significantly lower coverage. Both involve running representative code as a facsimile of the real workload execution. However, errors are often data-dependent [1], such that only a small fraction of the input space triggers them, and temperature/voltage variation causes errors to be intermittent [1], [70], meaning coverage is inevitably incomplete: Fleetscanner detects 93% of permanent faults within 6 months, and Ripple detects 70% over shorter timescales [6].

B. Heterogeneous Parallel Error Detection

To achieve acceptable power overheads, we can use ParaMedic-style [11], [12] heterogeneous parallel error detection. Thread-level parallelism that may not have even existed in the original run allows extreme efficiency. The fundamental idea (see fig. 1, *error-detecting capability*) is that, if a log tracks all stores the CPU performed, and all load values it observed, then computation of the checked workload/thread can be split up, and the repeat execution between each "checkpoint" can be overlapped. A new check can be started before the previous one in program order finishes because it is only dependent on the register files, loads and stores logged from the original execution. An induction-style approach can be used to detect matches: checkpoint N is correct provided checkpoints 1...N-1 are correct, all loads and stores were to the intended addresses, all stores match those of the original run, and the register file at the end of segment N matches the start of the original thread's execution of segment N+1.

In the original papers on the subject [11]–[13], each main core is surrounded by a sea of small, microcontroller-sized cores dedicated to error detection. This is energy efficient, since each checker core can be individually slow, and without data caches. However, if used in a system that does not always require error detection, the 25% [11] extra silicon is difficult to justify in terms of power/performance/area overhead (PPA). We could instead repurpose existing cores in a system, still utilizing heterogeneous parallelism for energy efficiency relative to lockstep, by running checking code across multiple existing smaller cores, as now featured in Arm's big.LITTLE [14], [71] systems, AMD's Zen4 and Zen4c cores [15], [72] and Intel's recent P- and E-class cores [73], or on multiple homogeneous cores running at lower clock frequencies.

C. Opportunistic Parallel Error Detection

Our goal is to find unreliable computation within servers both more quickly and at higher detection rates than software-only scanners [6]. Still, high performance overheads will detract from the core work of data centers: we need a spectrum of options. We can give full detection of any incorrect computation at any point, slowing down the system when there are too few idle resources to keep up with the checked workload. The same system should allow a coverage reduction to maintain performance when immediate detection is less important than avoiding any slowdown at all, or when all the heterogeneous resources of a system are fully utilised (fig. 1). This is similar to sampling-based lockstep [69], but without the need for identical hardware for the redundant execution, or synchronisation of cycle-by-cycle lockstep.

An implication of the server domain is that we only require error detection [11], rather than hardware rollback, as the additional performance overhead [12] is unlikely to be justifiable unlike in safety-critical scenarios [10]. The core goal [6] is to retire unreliable cores (rather than catching all soft errors, e.g., cosmic-ray bit flips, though our full-coverage mode will detect these). Software stacks in the data center are typically already fault resilient provided the fault is captured, as distributed nodes are assumed to fail constantly [74], [75][1].

If all we need is high coverage of hard or semi-hard errors (which have been observed to be the main blight in server-class systems [1]), as opposed to transient bit flips, we need not cover 100% of executed instructions. We can opportunistically use spare cores, or cores whose single-threaded throughput is too low for the task at hand, when they are available.

D. Implications for Checker Cores

If we are reusing spare server cores that will sometimes be better used for scheduling user code, rather than small, low-performance, dedicated fault-tolerance engines [13], the checker cores inevitably will not be as small or as parallel as

[1]If full correction and synchronous guarantees were needed in hardware, due to lack of fail-safety otherwise seen in today's data-center stacks [74], [75], we could use rollback and dynamic-checkpointing stategies [12]. These would add 1% overhead but change nothing else discussed in this paper.

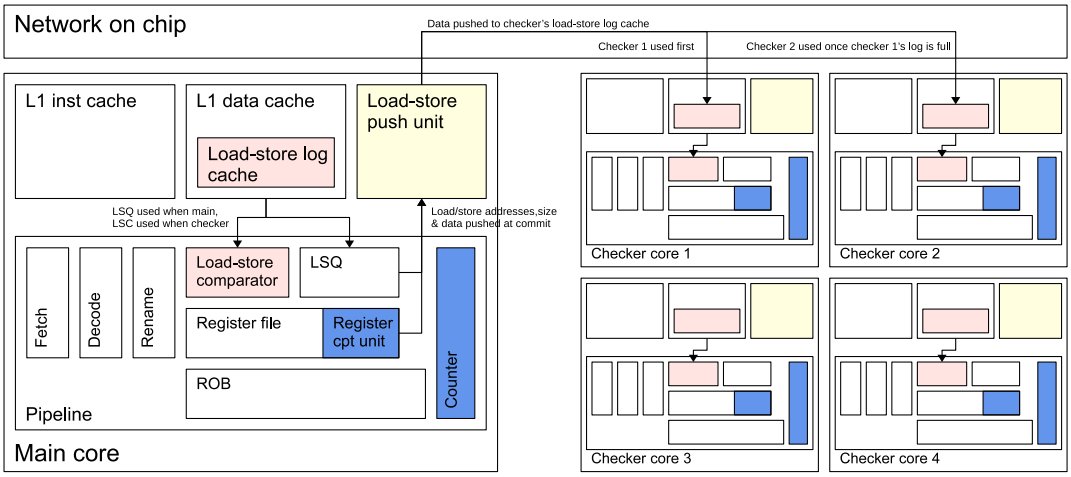

Fig. 2: ParaVerser adds several units to each core – a Load-Store Push Unit, a Register Checkpointing Unit, a Counter, A Load-Store Comparator (for accessing the Load-store Log Cache), and a microcode switch that triggers the LSC as appropriate. All cores feature the same modifications, as any core can be treated as either main- or checker-core, with smaller checker cores preferred if free. New units used main-core side in yellow, checker-core side in pink, and both sides in blue.

in previous work on efficient error detection [11], where 12–16 tiny checker cores were placed next to each main core. In our study, we focus primarily on heterogeneous big.LITTLE-style [71] microarchitectures since (i) this enables energy efficient error detection on little cores and (ii) future data-center SoCs are expected to contain heterogeneous cores [14], [15], [76], though ParaVerser can be also efficiently implemented on homogeneous cores, as we demonstrate in section VII.

If checker cores are each higher throughput, we will need fewer of them, even if each is larger in power and area. Since we are extending hardware with functionality not constantly used, the area impact must be minimal, reusing hardware where possible: for example repurposing existing SRAM cache memories instead of dedicated storage [11] for fault-tolerant logging, and existing network-on-chip (NoC) layouts instead of dedicated wiring [11] for forwarding intermediate results.

E. Why CPUs?

ParaVerser is a mechanism for detecting CPU faults. In a modern system-on-chip the CPUs do not collectively cover the entire chip area, nor do they perform all computation, with much area spent on caches and accelerators such as GPUs and NPUs [77]. We focus on CPU due to a combination of both benefit and opportunity. On benefit, the CPU is where much of the most error-intolerant work occurs: for example the kernel, and lots of control-heavy code where even small changes in input lead to drastic changes in output, as evidenced by the recent focus on CPU errors across hyperscalers [1], [2], [18], [78], [79]. On opportunities, caches can be covered by ECC, whereas CPUs cannot[2]. Since induction parallelism [11]–[13] allows the trailing run to be executed on more power efficient cores, there can be very small power and area overheads compared to the high-IPC cores used for maximum single-threaded throughput, which waste lots of energy on instruction-level parallelism. By contrast, GPU and NPUs do not aim for high single-threaded throughput, so a similar heterogeneity trick is unlikely to reduce overheads: we cannot make the trailing threads more efficient by making them slower and more parallel unlike on CPUs, so it is challenging to outperform lockstep-style homogeneous redundancy. Last but not least, the importance of CPUs is growing even in the AI era due to their wide availability and greater energy efficiency in some key LLM inference [80] scenarios compared to GPUs or NPUs. Several hardware companies are working to extend their CPUs to efficiently handle AI workloads [81].

[2]While controllers inside caches that actually do computation to e.g. choose tags cannot be protected by ECC, these are small enough that homogeneous replication (lockstep) of those structures comes at negligible cost (unlike the cores themselves which are large so homogeneous replication is expensive, as we explore in section VII-E).

IV. ParaVerser

Figure 2 shows ParaVerser's minor core alterations. These allow any core to behave as either a *main core*[3] or as a *checker core* — with the intent that checker cores will be more numerous, parallel and energy efficient. For cache- and main-memory errors we assume ECC [4] or parity is used.

- We describe how checker resources are managed and allocated differently between full-coverage and opportunistic modes (section IV-A). The former pauses main computation to let checker cores catch up if they are collectively slower, and the latter drops excess instructions from checking.
- We augment each L1 data cache to allow a checker core to store logged memory accesses and register checkpoints, for parallel replay of computation segments (section IV-B).
- We add a unit to directly *push* recorded loads, stores and other non-repeatable events over the existing NoC to an arbitrary checker core's logs, avoiding shared memory and coherence overheads (section IV-C).

[3]Without loss of generality, we refer to *a main core* having multiple checker cores, but there may be multiple main cores each with multiple checker cores.

- We then add a register-checkpoint unit (section IV-D), which generates start/end register checkpoints on the main core and stores and compares end register checkpoints on the checker core(s), and a load-store comparator (section IV-E), which compares memory addresses and stored data with logged versions from the load-store log cache.
- Lastly, we add a counter unit (section IV-F) to interrupt main and checker cores at identical instruction counts, to allow replay. Segments are split up via checkpoints generated in the register-checkpoint unit.

We also design new mechanisms to handle checker cores that are *themselves* out-of-order and/or superscalar, perhaps with entirely different microarchitectures from the main core (section IV-G). This requires support for speculation and reordering with respect to the main core's execution, while still observing the correct, and equivalent, logged load-and-store behaviour of the original run. We also improve the efficiency of the design by giving checker cores the ability to start execution before the main core has finished the relevant checkpoint (section IV-H) — and give constraints on load-and-store behaviour to avoid inconsistency between the two that may otherwise result. We then propose a method to reduce cross-core data transfer to avoid slowdowns on underprovisioned NoCs (section IV-I). Finally, we explain how multiprocess and multicore workload behaviour is handled (section IV-J).

A. Basic Operation

For a given main core, a free checker core is chosen. A copy of the register file is taken (section IV-D) and pushed to the checker core, which will execute from the same point. Loads and stores are logged by the main core and sent to the checker core for it to replay memory accesses. Once the Load-store log cache (section IV-B) is full, a timeout is reached or an interrupt received, an end checkpoint is taken and sent to the checker core for later verification.

As a main core deliberately has higher single-threaded throughput than a checker core, the previous checkpoint is fully verified only some time after the main core creates it. To avoid stalling, it parallelises the error-detection process by selecting a new checker core for the next interval while the previous checkpoint is still being checked. In full-coverage mode, if all possible targets are currently occupied with either error detection (e.g., because the available checker cores cannot collectively keep up with the main core), or are scheduled to run other programs, the main core then pauses execution. Once free resources are available, a new starting checkpoint is sent to a new checker core, and computation proceeds.

In opportunistic mode, if there are no checker resources remaining, register checkpointing and logging are briefly switched off. While previous segments continue to be checked, the current checkpoint is not forwarded on, and the main core proceeds without checking to avoid performance hit. Once a previous checkpoint has been checked, freeing the checker core, the main core will immediately take a new checkpoint to start getting checked again from this point.

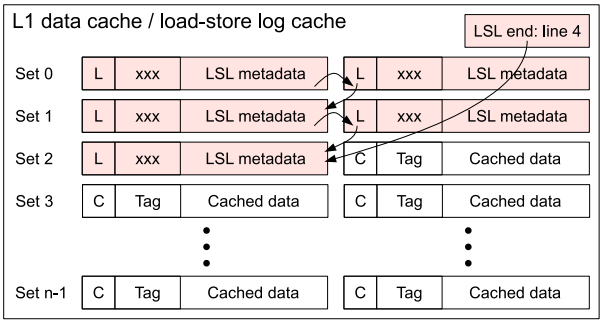

Fig. 3: Cache layout when used as a LSL$. Cache lines become replaced with load-store log (LSL) entry metadata starting at index 0 – with the load-store-log end register indicating the final valid entry. Cache lines following this retain valid data.

We let the operating system decide, based on its current load, which CPUs it allocates as checker cores and which it allocates as main cores[4]. Preference for allocation as checker cores is given to idle cores, and lower-performance cores if available, since checking does not require high single-threaded performance. A core can be switched back from a checker to a main core at the end of each checkpoint[5] if the operating system decides there are currently more checkers than required or it needs more main-core compute.

B. Load-Store Log Cache

Previous techniques (e.g., Ainsworth and Jones [11]) use a dedicated SRAM load-store log, which introduces a memory-storage overhead. In contrast, we lightly modify and repurpose the data cache that will already be available on a general-purpose core, using it to store data to regenerate computation and verify correctness (except in Hash Mode, section IV-I, when metadata for verification are not stored in cache). We call the new structure a Load-Store Log Cache (LSL$).

Data stored in the LSL$ for computation replay contain the loaded data and the values of other non-repeatable instructions, such as store conditionals, timers, reads/writes to system registers, and random-number generators. This allows exact replay regardless of any multicore communication in between. Data stored for verifying correctness include load/store addresses, sizes and stored data. A typical LSL$ entry consists of a 7B address followed by a 1B size and a variable-length payload for data rounded to the nearest 8B (first loaded then stored data in cases where both are required, e.g., a SWP in Arm)[6]. The entries are first filled in the LSPU (section IV-C) in-order at commit time from a main core and will be interpreted as the same sequence[7] on the checker. Then cache lines of entries (a

[4]The evaluations in section VII assume a fixed set of checker cores within each individual experiment, to allow for simple comparison.

[5]Since checkpoints are limited in size (max. timeout instructions), there is no resource-starvation issue from preventing pre-emption of checkpoints.

[6]In Hash Mode (section IV-I), we only store the payload: all other metadata (used only for verifying correctness) is hashed into the SHA-256 checksum.

[7]The actual access order may deviate from this under out-of-order or speculative execution within checker cores, as handled in section IV-G, but logical in-order behaviour is preserved.

512-bit cache line stores 4 entries with a typical 64-bit result each) are pushed over the NoC to the LSL$.

The structure of the LSL$ is shown in fig. 3, where each cache-line tag gains one extra bit to indicate whether the data stored is from a log (L) or a cached copy of main-memory data (C). When a core starts being used for checking, its cache starts being repurposed as a linear log[8] rather than a content-addressable memory of addresses[9] . We start filling from the first index and set of the checker core's cache, evicting the cache line in place (if valid and not already a log entry). A word stored in a new Load-Store Log End register indicates which line is the current end element.

C. Load-Store Push Unit

We add a Load-Store Push Unit (LSPU) to the cores, so that load-store log (LSL) entries can be sent between main and checker cores over the generic NoC. Unlike in previous heterogeneous error detection techniques [11]–[13], ParaVerser requires all-to-all communication between cores (any core can check any other). The overheads of this cross-core communication are mitigated by buffering a cache line's worth of LSL entries locally at each main core in the LSPU, and because LSL$ entries are treated as scratch memory rather than coherent traffic, they can be sent direct instead of going via a directory or the last-level cache (LLC). Unless an entry is itself larger than a cache line, entries that cannot fit in the remaining space of the current line are put into the next line.

To allow the microarchitecture of big and little cores to diverge, LSL$ stores data in ISA format. This requires fusing together data from multiple micro-ops at pipeline commit. For each load/store micro-op at commit, the main core continues to update data in the same LSL entry in the LSPU when accesses are from the same instruction. This merged entry covers all loaded data starting from a single base address, followed by any stored data from the base address[10]. Upon LSL$ access, the accessed address and size are compared against the range between the address and size in the LSL$ entry, and the address is used as an offset into the LSL$ entry data segment to retrieve, or check, corresponding data.

The data (section IV-B) to be pushed to the log are accessed from the core's load-store *queue* (LSQ) at commit time[11]. To avoid limiting coverage, we make minor changes to the LSQ: any error in a store that reaches memory must also reach the checker core. Conversely, any error in the loaded value must not reach the checker core, so that at least one receives the correct value. For loads, ECC or parity bits from the cache, whichever are used in the system, are forwarded to the load queue, and checked before being forwarded to the LSPU. For stores, ECC or parity bits are generated before the store propagates to both the local cache and to the LSL$. No common-mode errors between the main and checker cores are propagated; parity and ECC are not needed for the NoC communication, the checker core, or the main core, or any register files, as errors in these components are isolated to either the main core or checker core.

The LSPU is the same size as a cache line, the NoC width, or the largest possible LSL$ entry generated by a single instruction of the target ISA (whichever is larger), and is pushed to the NoC when either it is full or an end checkpoint is taken (and thus we change checker core). Intermediate states may exist in the LSPU that violate ISA compatibility, if not all micro-ops for a macro-op have yet committed, but all data pushed over the NoC obeys ISA compatibility.

D. Register Checkpointing Unit

The register checkpointing unit (RCU) is used to take start and end register checkpoints on the main core, and to store end register checkpoints on the checker core. At the start of a checkpoint, the RCU takes a copy of the architectural register file, forwards it via the NoC to the chosen checker core, which updates its register file and begins checking. At the end of a checkpoint, the main core's RCU forwards a new copy of the architectural register file to the chosen checker core's RCU, and updates the newly allocated checker core if available. Finally, once a checker core is interrupted by the instruction counter, the architectural register file is compared against the RCU copy. While system-visible state is checked via loads and stores in the LSL or the hash value in Hash Mode (section IV-I), register-file checks at the start and end of each checkpoint are also needed to verify the full program sequence correctness, via induction [11].

E. Load-Store Comparator

The Load-Store Comparator (LSC) compares the address and size generated by the checker core with the ones stored in the log for each load and store. For stores, it also compares the logged value with the checker core's result. For loads, this occurs out-of-order (section IV-G): when a LSL entry is accessed, the data payload is copied into the load queue, and the address stored in the load queue is compared against the value from the LSL$. For stores, this occurs at commit: when a store is committed, the LSL$ entry is compared against the address and data in the store queue. To avoid slowdown, there is one comparator for every load or store unit in the core.

F. Instruction Counter

The counter is used to precisely match the checkpoint end between main and checker cores. On the main-core side, checkpoints are generated if either (i) the LSL$ becomes full, (ii) there is an interrupt or context switch, or (iii) a 5000-

[8]A main core's cache is unaffected; it does not access logged entries itself.

[9]A similar form of partitioning [82], [83] is already used in Arm L1 caches as of the X4 [84], for temporal prefetching [85]. Since our use involves direct indexing, with no tag comparisons to find the correct value, it is far simpler.

[10]The exception to this format is when a single instruction has more than one base address, such as a scatter or gather. In this case, microarchitectural invariance is achieved by storing each address, associated data size, followed by associated data, in sequence, with lowest address first.

[11]Previous work [11]–[13] has a more elaborate *load-forward duplicator*, which stores entries per-ROB index. With deeper knowledge of core microarchitecture here, we observe it is much simpler to reuse the LSQ, which stores the load addresses and data we need already.

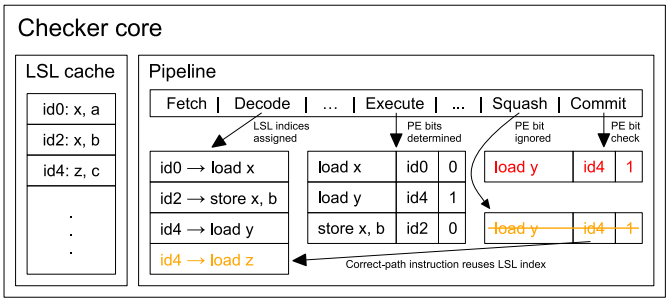

Fig. 4: Indices are assigned speculatively at decode. In this example, 3 instructions (I1, I2, I3) are assigned indices 0, 2 and 4 respectively in program order (for offsets of 0, 16 and 32 bytes into the LSL$). In the checker core's out-of-order backend, the instructions are reordered such that I3 occurs before I2. Still, they access the correct entries as the LSL is accessed via index rather than sequentially. I3, a load to y, triggers an error because the value in the LSL$ is a load to z. This sets the precise exception (PE) bit in the reorder buffer for this instruction. If I3 becomes non-speculative and retires, it has requested different data from that loaded in the original run, so an error is reported. If I3 is squashed, the load was a misspeculation, so the index is reset and the new, correct execution path accesses the same log entry instead.

instruction timeout [11] is reached[12]. At this point, a copy of the architectural register file is taken by the RCU, and forwarded to the checker core. On the checker core, we finish the check, and compare register files, at the same committed-instruction count as the time of checkpoint on the main core.

G. Speculative Out-Of-Order Checker Cores

The LSL is filled in program order, at commit time of the main core, as in previous work [11]–[13]. However, prior work relied on in-order access of the LSL for checking, limiting the checkers to the simplest in-order cores, where misspeculations never reach the data path and memory accesses could not be reordered. This is a false assumption for typical programmable server cores, even the smallest in our evaluation.

To fix this, we use an indexed-access scheme for the LSL$. Out-of-order checker cores no longer access the log in sequential order. When the checker cores decode a load/store, we increase a speculative counter at the in-order front-end based on the size of the expected LSL$ payload (section IV-B), so that the index will point to the appropriate entry in the LSL$ in the program order (see fig 4). For loads, this speculative index follows the instruction into the load queue, sitting in the data-payload field until it is overwritten by the access returning data. For stores, it is not stored explicitly and is instead regenerated at commit time, when the access to the log proceeds. Where a load/store instruction is broken down into micro-ops, these micro-ops share the same index.

Due to the speculative nature of the index, an access to the LSL$ may not match the indexed entry despite no error having occurred: the misspeculated instruction will be squashed, and the accessed entry is intended for an instruction that will be executed on the return to correct execution. Thus suspected faults must be handled as precise exceptions: we record errors on LSL$ access but do not raise them until commit.

Speculative indices must be adjusted when instructions are squashed to match the commit order. We handle this by deducting from the frontend's speculative index on every squashed instruction. Also, this index is reset to 0 when the checker starts checking a new LSL$ segment/checkpoint.

H. Eager Checker-Core Waking

In previous work [11]–[13], checker cores are only woken once a checkpoint has finished, to ensure consistency of execution. If checker cores are sufficiently simple to be almost free, this is sensible, but if checker cores are the size of a conventional core, this wastes resources by meaning at least one core is always stalled waiting for a checkpoint to finish.

A checker core can commence early as long as it never executes ahead of the main core, and so never reads an invalid LSL$ entry, and cannot execute instructions never executed by the main core, due to timeout being reached or an interrupt (section IV-J). To enable this while ensuring matching behaviour and preventing the checker core from running ahead, we use the LSL$ as a limiter. A checker core cannot execute any instruction past the last LSL$ entry currently pushed to the checker core. If it tries to do so, and the RCU checkpoint has not yet been set, the checker core sleeps until a push of a cache line into its LSL$ or the RCU checkpoint being set.

A memory-access instruction attempting to read the last valid entry[13] in the LSL$ causes all following instructions to squash (except any micro-ops as part of the same macro-op). When a new log cache line arrives or a checkpoint register file is received to indicate the end of the checkpoint, fetch is restarted from the first squashed instruction. If this access of the last valid entry *itself* is squashed, instruction fetch is similarly restarted, with the core returning to sleep on any subsequent attempt to read the final current LSL$ entry.

I. Hash Mode

ParaVerser uses considerable NoC traffic to send LSL$ entries from the main core to the checker core. To limit this, we also provide a *Hash Mode* where only data required to reproduce execution (e.g., loaded data) are recorded in LSL$ entries and transferred over the NoC (entries are still stored in-order contiguously). Data used only to verify correctness (e.g., addresses and stored data) are checksummed, and only the hashed value is transferred over the NoC and compared at the end of a checkpoint. Hash Mode reduces the traffic by

[12]ParaVerser will fill the entire L1 unless it reaches the 5000-instruction timeout first – whenever a checkpoint finishes (which is when a workload becomes eligible to context switch in and take the core), all of this space is freed. So ParaVerser using this space is no different from any other thread filling the cache with data. A checker thread itself needs no data cache, as it cannot read data, so there are no tradeoffs in cache-space usage. 64KiB shared with the data cache is more than ample to make checkpoint frequency rare and thus inexpensive, whereas 3KiB of dedicated SRAM [11] is not, and the cache is otherwise unused by a checker thread.

[13]Or beyond the last valid entry in an out-of-order core, in which case the memory-access instruction itself is also squashed.

50% for loads and eliminates it entirely for stores, but the reliability will depend on the hash-function properties. Hash functions that cannot detect repeated error on the same bit or reordering should be avoided — here we use SHA-256 [86].

Hash Mode requires several modifications to the mechanisms mentioned previously. In Hash Mode, the LSL$ entries only contain data required to reproduce execution in commit order, so the out-of-order cores' speculative index also only increments when the instruction has such data. For micro-ops, the offset into the entry is calculated at decode time and stored alongside the index until the LSL$ is accessed. Both the main and the checker cores use the LSPU to buffer data to be digested for the hash calculation at instruction commit to maintain the access order. The LSC is no longer used by the checker core to detect LSL$ access errors. Instead, the hash value is calculated in the RCU, and is sent along with the register checkpoint to the checker's RCU for comparison.

J. Multiprocess and Multicore

To avoid the need to replay interrupts at the same time between main and checker cores, register checkpoints are taken whenever an interrupt occurs. This is also true for context switches: as a result, each register checkpoint is associated with only one process. Checks from multiple processes on one main core may be checked on multiple checker cores concurrently, just as multiple processes on multiple main cores may be checked concurrently. If an error occurs in any particular process's checkpoints, the error is raised as an exception to that process. ParaVerser is not an error correction system, as it incurs latency in between execution and checking to achieve thread-level parallelism, and so software must clean up after itself if errors are discovered (section III-C).

ParaVerser's logging extends to multicore shared-memory workloads with no modification [12]. Because the exact loads and stores seen by the main core are then propagated to checker cores, any resulting cross-thread communication is mimicked and checked exactly. Because load-store logging records loads at the time the first, main-core execution(s) occurred, any race conditions, and cross-thread communication more generally, between threads replay exactly as they did on their first run (contrast e.g. with a scenario where the checker threads truly repeated the loads from memory, where they would be impacted by what happened in what order). Likewise, this means synchronisation between threads only affects what is written to the load-store log, rather than having a direct impact on e.g. checkpoint lengths (section IV-F). In terms of checker-core allocation, we treat each main core as a separate checking task, in turn split up across many checkers.

V. Sphere of Replication

ParaVerser is a redundancy mechanism for computation, and thus the sphere of replication is the core itself, with the boundary at the LSQ where the contents are replicated and transferred to the checker core through the LSL. Caches are outside the sphere of replication and need parity or ECC

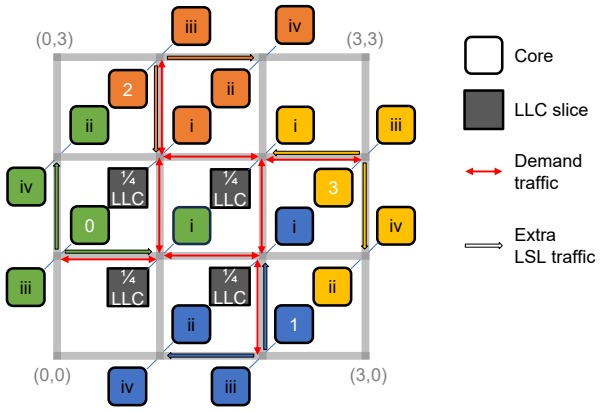

Fig. 5: NoC-tile layout, where arabic numbers denote main cores and roman numerals denote the checker cores in the same colour as the main cores they check.

to ensure their correctness, along with redundancy on any computation inside the cache system (such as for coherence).

Checker-core execution does not repeat data-address translation and assumes that the load data recorded in the LSL is correct; additional redundancy in the page-table walker [12] and LSQ should be added if coverage of every transistor of the core is desired. In the latter case, this involves both propagating and checking parity bits on data before they are sent to the LSL, as well as redundancy to catch errors from faults in the LSQ logic itself (e.g., an access-order violation misdetected due to bit flips [87]).

Under full-coverage mode, ParaVerser detects both hard and soft errors from compute in the system. Opportunistic mode only targets hard errors that can be eventually detected, but will also detect soft errors in the checked segments. Since we do not distinguish between hard and soft errors, the operator will need to run their own tests after we detect an error to determine whether the core has a hard fault and needs retiring. If more precise forensics are desirable, our starting register checkpoints allow repeat replays to identify culprits, at the expense of a further 776B overhead per core. Errors that do not change execution will still be detected if the contents of the LSL, the hash value in Hash Mode, or the register values at the start or end of a register checkpoint are changed, and we cannot directly distinguish whether errors are from the main or checker core. These can be considered as false positives since the detected errors do not affect the main core's execution. However, these still represent real errors that occurred somewhere in the system; they thus still help in retiring faulty cores early if they were hard errors. In Hash Mode, we use SHA-256 hashing since collision of hash values produced by values with and without error is highly unlikely, with 128-bit security of collision resistance [88].

VI. Experimental Setup

To evaluate ParaVerser, we ported the ParaDox simulator [13] to gem5 v22.0.0.1. We added the new mechanisms described in section IV, and designed CPU models for high-

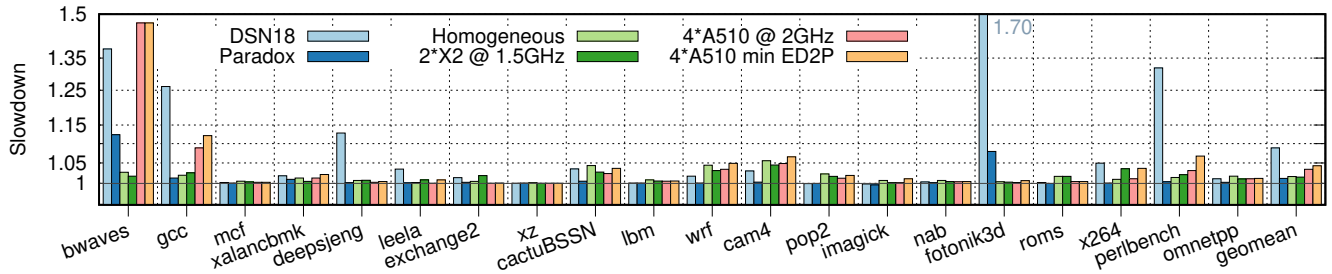

Fig. 6: Slowdown of main core (3GHz X2) with different checker-core configurations, with full-coverage mode enabled.

TABLE I: Core and Memory Experimental Setup.

Big Cores	
Core	5-Wide, out-of-order, 3GHz in Main Mode (up to 3GHz in Checker Mode)
Pipeline	288-Entry ROB, 120-entry IQ, 85-entry LQ, 90-entry SQ, 150 Int / 256 FP registers, 2 Branch ALUs, 2 Simple Int, 2 Complex Int, 4 FP/SIMD, 1 Load-Only, 1 Load-Store
L1 ICache	64KiB, 4-way, 2-cycle hit lat, 16 MSHRs
L1 DCache	64KiB, 4-way, 4-cycle hit lat, 16 MSHRs
L2 Cache	1MiB, 8-way, 9-cycle hit lat, 32 MSHRs
Branch	64KiB MPP-TAGE
Reg. Checkpoint	8-Cycle latency, 5,000-instruction timeout
Little Cores	
Cores	3-Wide, in-order, up to 2GHz
Pipeline	16-Entry LSQ, 1 Branch ALU, 3 Int, 1 Div, 2 FP/SIMD, 1 Load-Only, 1 Load-Store
L1 ICache	32KiB, 4-way, 1-cycle hit lat, 12 MSHRs
L1 DCache	32KiB, 4-way, 1-cycle hit lat, 12 MSHRs
L2 Cache	256KiB, 8-way, 9-cycle hit lat, 16 MSHRs
Branch Predictor	8KiB MPP-TAGE
System	
L3 Cache	8MiB, 8-way 25-cycle hit lat, 48 MSHRs
Memory	DDR4_2400_8x8
NoC	2D Bidir. Mesh, 256 bit width, 2GHz
slowNoC	2D Bidir. Mesh, 128 bit width, 1.5GHz

performance server-style Arm[14] cores (table I), based on publicly available information on both big Cortex-X2 [90]–[92] and little Cortex-A510 cores [93], [94][15], which form the basis of Arm's heterogeneous data-center Neoverse V2 and E2 [14]. Our main cores are always out-of-order at 3GHz; we run varying numbers and types (big, little) of checker cores at various clock frequencies. Prior works [11], [13] used the generic MinorCPU model from gem5 in their evaluation, which, for example, models an overly simplistic FPU with unrealistic 6-cycle latency for all floating-point operations when instructions such as divisions can take up to 22 cycles [93], [95]. Even for integers, it lacks modeling of variable latency for operations sharing the same functional units, unlike in gem5's HPI core, which we modified for our A510 model. This means that the core model used in prior works does not provide a reasonable comparison against our more detailed X2 and A510 core models. Therefore, to compare against prior works DSN18 [11] and Paradox [13], we also model dedicated checker cores based on Cortex-A55 [95] and limit them to be scalar to emulate the performance of Cortex-A34/35 [96] cores (due to the lack of documentation): the smallest in-order Cortex-A processors supporting AArch64 [97].

We evaluate on the SPECspeed 2017 benchmarks. Statistics are taken from detailed simulation for 1B instructions after fast-forwarding for 10B unless stated otherwise. To explore data-oriented workloads, we also run GAP [98], skipping initialisation, and for parallel workloads we run PARSEC on simmedium to completion [99]. Slowdowns are presented relative to a baseline without checking. When evaluating full-coverage mode, we assess performance overhead. With opportunistic mode, we assess checking coverage as well.

We model NoC latencies by feeding the gem5 network parameters into an MM1 queueing network model of a 2D mesh. Figure 5 illustrates the layout used for our experiments, with main cores denoted with Arabic number 0–3, and checker cores for the same-coloured main core denoted with Roman numerals i–iv. The NoC is configured with 256-bit width and operates at 2GHz unless stated otherwise, similar to ARM Neoverse CMN-700's mesh [100]. The four crosspoints in the middle each have an LLC slice and a core attached; each LLC slice is assumed to provide $\frac{1}{4}$ of demand data for each main core. Except for the four corners, each crosspoint has two cores attached. We chose cores on crosspoints without LLC slices attached to be used as main cores since this case is more common in our layout, and the cores adjacent to the main core are used as its checker cores. When using heterogeneous main and checker cores, this layout represents a tiled system with both big and small cores distributed throughout the mesh, rather than clustered in homogeneous groups[16]. Evaluations

[14]While we explore Arm here, the same concepts apply to other ISAs such as x86. One minor issue we foresee is that there may be some instructions in x86 where a single macro-op, such as REP MOVS, processes more data than can fit into the LSL$. In those cases only, micro-op behaviour must match more closely between big and little cores, rather than only macro-ops needing to match for ISA compatibility. X86's ISA being more complex possibly puts a limit on how small/power-efficient checker cores can be, but we still expect a good energy-efficiency improvement from heterogeneous checking. Likewise, though Intel's standard gen-14 cores are also heterogeneous [73] cores, our own testing [89] revealed that the little cores do not have a separate voltage domain from the big cores unlike Arm/Apple systems where we found they did, limiting energy-efficiency gains, but that is likely to change in future.

[15]In the rest of the paper, we call the big cores based on Arm Cortex-X2 "X2" and the little cores modeled on Arm Cortex-A510 cores "A510".

[16]If cores were more homogeneously clustered, we expect checker-core traffic to have higher average hop counts, and thus conflict more often with latency-sensitive LLC accesses. We expect minimal difference to the presented results for other minor variations in structure, such as further distribution of the LLC, which would move the baseline traffic around but not typically increase it beyond the contention caused by using only checker core i.

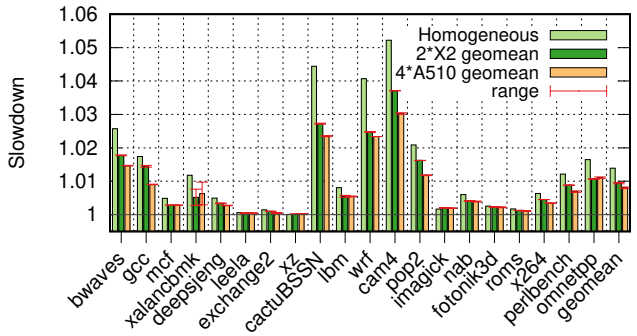

Fig. 7: Slowdown of main core (3GHz X2) with different checker-core configurations, with opportunistic mode enabled. Error bar shows the range of slowdown with different configurations of the same type of checker cores.

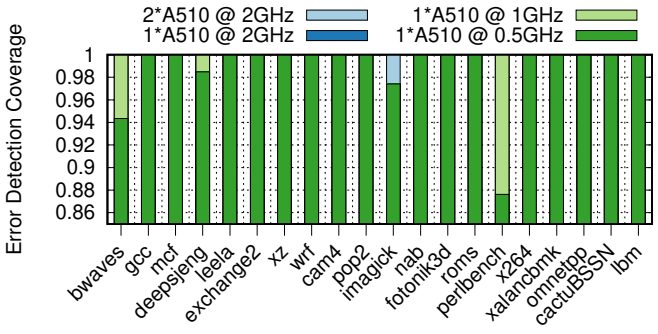

Fig. 8: Error detection coverage under opportunistic mode with different checker-core configurations. Legend shows minimum required configuration to cover such portions of errors.

with one main core use core 0 as the main core; those with two main cores use cores 0 and 1. When selecting checker cores, checker core i, which incurs contention on demand traffic and results in worse performance overhead, is used first. If more checker cores are needed, cores ii–iv are used. Unused main and checker cores generate no traffic. We backpropagate the observed average latency from additional LSL$ traffic into gem5's LLC access latency to estimate overhead, and explore NoC-bandwidth impact in section VII-D.

VII. Evaluation

A. Full-Coverage Mode

Figure 6 shows the slowdown of the main core with various checker-core configurations in full-coverage mode (all dynamic instructions are checked), compared to prior works DSN18 [11] with 12 and Paradox [13] with 16 dedicated checker cores. By configuring the checker core as identical to the 3GHz main core in a homogeneous system, the checker core keeps up with the progression of the main core with a geomean slowdown of 1.6%. The DSN18 configuration shows a geomean slowdown of 9%, insufficient to keep up with our X2 main-core performance. While Paradox with 16 dedicated checker cores shows only 1.2% slowdown, this is at the expense of 35% area overhead (section VII-E), limiting the silicon directly usable for compute. ParaVerser provides checking capabilities with cores already in the system, and does not affect the performance when checking is disabled.

Employing 2 X2 checker cores operating at half frequency (1.5GHz) results in almost the same slowdown as a homogeneous system. While the LSL traffic contending with demand traffic on the NoC is reduced, the benefit is balanced off by the slight extra performance overhead coming from unshared components, such as icache prefetch and branch-prediction training, resulting in slightly more misses and mispredicts.

With 4 A510 cores operating at 2GHz as checker cores, the slowdown is 3.4%. The checker cores' collective performance typically matches the main core except in bwaves. The large number of fdiv instructions in bwaves and the large difference in floating-point division performance between the big X2 core [90] and small A510 core [93] results in the checker cores at their worst struggling to keep up with the main core at its best. Via DVFS, we also vary the A510 checkers' frequency from 2GHz to 1.4GHz and the voltage accordingly to find the best ED2P point per benchmark. Slowdown is still only 4.3%.

Next we consider the causes of overheads:

Register Checkpointing While in previous work [11] register checkpointing accounted for a significant overhead, by delaying the main core's commit to copy the register file, the overhead in ParaVerser is negligible. By repurposing a conventional core's cache, we have much larger LSL (64KiB versus 3KiB [11]), and so checkpoints happen less frequently.

Stalling Overhead In full-coverage mode, the main core must stall until a checker core is available when checker cores cannot keep up with it. This is the main factor as checker cores reduce in clock frequency, number, and out-of-order ability.

Instruction Fetch While checker cores never cause accesses to main memory or shared caches from data, as their loads and stores are served via the LSL$, they still access instructions through them. This causes a slight contention effect in both full-coverage and opportunistic mode, particularly in workloads such as gcc, which have frequent L1 icache misses.

NoC Overhead Loads and stores are logged and forwarded to checker cores over the NoC in groups based on bus width (section IV-C). While this causes no significant latency for checker cores, which are pushed messages instead of requesting them, it causes secondary contention effects for other requests on the same network, particularly LLC accesses. We investigate the impact of this further in section VII-D.

B. Opportunistic Mode

Figure 7 shows slowdown with the same checker-core configurations as in section VII-A, but using opportunistic mode[17], which reduces coverage instead of stalling when out of resources. Unsurprisingly, overhead is lower than in full-coverage mode at 1.4% geomean slowdown for homogeneous and less than 1% for 2 X2 or 4 A510s. Overhead is mostly from NoC contention, thus is flat regardless of core frequency.

[17]Results with the same type of checker cores are similar, hence we only show the geomean and range. Checker core configurations with 2*X2 cores include 2*X2@1.35GHz and 2*X2@1.5GHz; with 4*A510 include 4*A510@1.6GHz, 4*A510@1.8GHz and 4*A510@2GHz.

To investigate opportunistic mode's hard-error coverage, we inject hard errors based on standard models from the literature [53], with various checker-core configurations, in fig. 8. Since the error detection is symmetrical, to avoid injected errors impeding the main core's execution, errors are injected on the checker core. We model the hard error as a single-bit stuck-at error, and inject the error on the output value from functional units modeled in gem5. Errors are injected to the output register for instructions using integer ALUs or FPUs and to the load/store addresses in LSQ. For ALUs and FPUs, a random bit in the output of a random functional unit is stuck at 1 or 0 to model an error in the logic rather than in the registers. Errors are injected depending on the type of operation and which functional unit is used: when there are multiple functional units available for an instruction, errors may not be injected depending on which unit is used. Under full-coverage mode, we injected errors for 10M instructions, and saw that 76% of injected errors were detected in full-coverage mode, and the remaining errors were (correctly) masked since they did not change execution. In fig. 8 almost all non-masked injected errors are detected with only one A510 running at 500MHz within 100M instructions. For most workloads this configuration detects all effective errors; only bwaves, deepsjeng, imagick and perlbench have lower rates of 87-99%. All except imagick achieve 100% with one A510 at 1GHz, and imagick achieves 100% with two A510s at 2GHz[18].

We also found that the the run-time instruction coverage, or proportion of executed main-core instructions that are checked, was high in opportunistic mode given sufficient checker cores. With a 3GHz X2 checker core, the geomean coverage is as high as over 98% with negligible performance overhead. With a lower checker-core frequency of 2.7GHz, it is reduced to 94%. With 4 A510 checker cores, the geomean coverage is 97%, 96% and 95% with their frequency at 2GHz, 1.8GHz and 1.6GHz respectively. Similar to the full-coverage slowdown, the run-time instruction coverage of bwaves is significantly lower than other benchmarks, at only 71% even with 2GHz A510s, due to its abundance of floating-point instructions.

C. Data-Oriented, Parallel and Multi-process Workloads

To show broader server-side workloads, we also look at the GAP suite [98] of graph workloads. GAP is so memory bound that even a small number of checker cores can keep up with the main core; fig. 9 shows that even in full-coverage mode 2 A510s are sufficient except for PageRank. ParaVerser also handles parallel workloads. Figure 9 also shows slowdown of ParaVerser in full-coverage mode on 2-threaded PARSEC [99]. Though PARSEC is not as memory bound as GAP, only 7.6% slowdown occurs when using 3 A510s per main core.

[18]The detection rates are high enough for most workloads that we envision using time-based sampling [69] in addition to heterogeneity would further increase efficiency, while keeping high hard-errors coverage. Still, calculating the precise proportion of compute that must be sampled to achieve high coverage probably highly depends on the exact fault behaviours of particular cores on particular silicon, so we do not directly investigate them here.

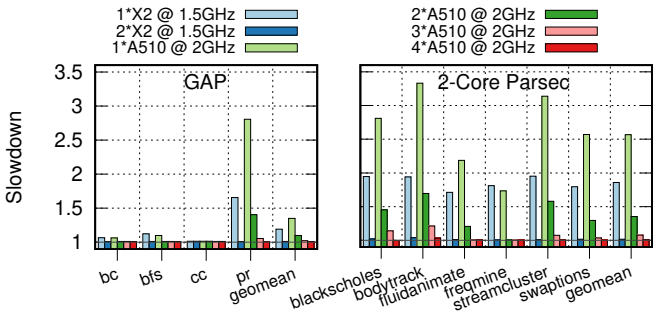

Fig. 9: Slowdown in full-coverage mode with varying checker-cores per main core, for GAP [98] and PARSEC [99].

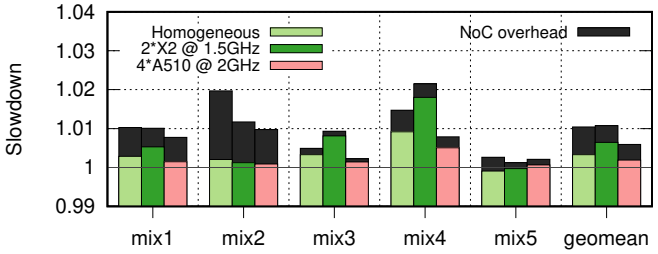

Fig. 10: Slowdown in full-coverage mode with varying checker core configurations for 4-core multi-process SPECspeed 2017. Coloured bars show overhead without LSL NoC-traffic impact.

To evaluate the impact of ParaVerser on multi-process workloads, we ran random mixes[19] of SPECspeed 2017 benchmarks on 4 main cores. We simulated for over 1 billion total instructions across 4 cores combined (after fast-forwarding for 10 billion instructions on the fastest core), with at least 250 million instructions on each core for mixes 1-4 and at least 100 million instructions on each core for mix5 due to simulation-time constraints. Figure 10 shows the slowdown on the total CPI with different checker core configurations. While the additional LSL traffic from one process contends with the demand traffic for other processes, the overall performance overhead is small with a geomean of only 1% with a homogeneous checker core or 2 X2 checker cores at 1.5GHz configurations, and less than 0.6% for 4 A510 checker cores at 2GHz.

D. NoC Sensitivity Study

To measure the impact of LSL traffic, we perform a sensitivity study with the checker cores configured at the highest frequencies and a slower NoC configuration of 128-bit width and 1.5GHz frequency. We also show the impact of enabling hash mode (section IV-I) on this slower NoC. Figure 11 shows that some benchmarks suffer significantly, resulting in a geomean overhead of over 15%. With hash mode reducing the LSL traffic by at least half, the NoC pressure is greatly alleviated, bringing down the geomean overhead to be within 0.8% of that of a faster NoC (256-bit width and 2GHz clock rate), which shows 1.5% NoC overhead for homogeneous checker cores and less than 1% with heterogeneous cores.

[19]mix1: bwaves, gcc, mcf, deepsjeng. mix2: cam4, imagick, nab, fotonik3d. mix3: leela, excahnge2, xz, wrt. mix4: pop2, roms, perlbench, x264. mix5: xalancbmk, omnetpp, cactuBSSN, lbm.

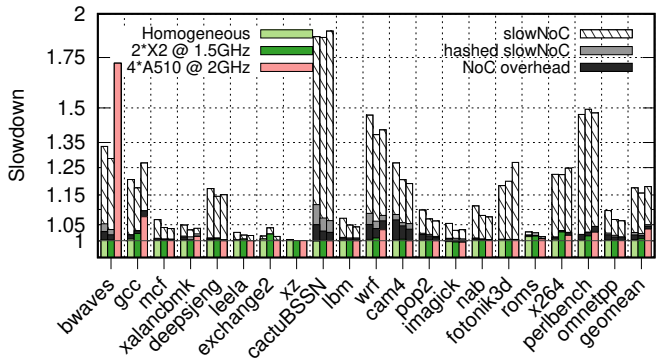

Fig. 11: Comparison of using a slower NoC configuration with hashed and non-hashed LSL at the highest core frequency, over 100 million instructions in full-coverage mode. Coloured bars show overhead without NoC impact.

E. Power and Area Overheads

ParaVerser adds only 1064B of storage overhead per core: 48B for a 2-wide LSC, 2 parity bits for each load- and store-queue entry (if not already present), 16 bits each for the front- and back-end LSL$ indices, 512 bits (or a cache line) for the LSPU, 1 bit per cache line in the LSL$, 13 bits for the instruction timer, and 776B for the RCU. Prior work [13] estimated the area overhead of dedicated checker cores to be less than a third of 2014's Cortex A57, but the dedicated checker cores' area were estimated based on RISC-V cores and compared to an ARM main core, despite both our ParaVerser and prior heterogeneous error detection techniques [13] requiring same-ISA, meaning for a fair comparison, all cores used for area estimation should be Arm in both cases. We re-evaluate this based on pixel count from die shots of the X2 and A510 cores [77], giving around 2.43mm^2 and 0.44mm^2 respectively with 4LPE Samsung technology. By extrapolating [101], [102] A35 sizes [96] from 28nm TSMC, we estimate the area of 16 A35s to be around 0.84mm^2, resulting in 35% area overhead.

While checker cores are repurposed from existing compute units rather than added new, using checker cores inevitably uses more power than leaving them idle. We examine the overhead via McPAT [103] configured at 22nm. For 4 A510 checkers running at 2GHz, we saw energy overheads (static and dynamic) of 49% geomean relative to a baseline with all checker cores power gated. For 2 X2 checkers at 1.5GHz, we also saw energy overheads of 45%, and for 1 X2 checker at 3GHz (homogeneous, so comparable to dual-core lockstep [10], [45], [46]) we saw energy overheads of 95%. By varying the frequency and voltage of 4 A510 cores, to reduce from 2GHz when possible, we can reduce further: an ED2P-minimal configuration with 4 A510s gives 29% energy overhead at 4.3% slowdown (versus 49% energy and 3.4% slowdown at full speed). In comparison, for prior work's dedicated checker cores [11], [13], we saw 25% energy overhead. Considering the extra 35% area overhead, this does not provide much energy savings compared to using the 4 A510s already in the system with ED2P-minimal configurations.

F. Compute Opportunity Costs

An alternative perspective is the extra compute that could be performed on cores used for checking. For single-threaded workloads, including most applications within SPEC, this is irrelevant: programs are not faster on multiple cores. Intuitively for parallel workloads, repeating all computation would result in a 2× slowdown. However, even parallel applications scale less well than checking, are more memory-intensive, and cause contention that slows down all cores. For GAP on 1 big core and 2 little cores[20], we see just a 1.52× speedup compared to 1 big core alone: the same number of little cores achieves full-coverage checking at 10% performance overhead. Likewise for PARSEC on 1 big core and 3 little cores, only a 1.44× speedup is observed, versus 7.6% overhead for the same setup with little cores devoted to checking. This compares favorably with a homogeneous setup, where using two big cores results in a 1.9× and 1.8× speedup for GAP and Parsec, respectively.

More generally though, there is no "free lunch" when it comes to duplicate execution, even if ParaVerser allows the use of highly power-efficient checker cores. Some systems will have all of their free resources used by programs carefully crafted to use every resource, and full-coverage mode will always come with compute-opportunity-cost overheads in such scenarios – in theory, with a worst case of 2× if all resources were perfectly utilised, though we do not see that in real workloads here. At this point, either more (power-efficient heterogeneous) resources would have to be added for fault tolerance, or coverage of only hard errors, via opportunistic mode, must be accepted – with varying guarantees depending on what limited resource can be spared (section VII-B).

VIII. Conclusion

ParaVerser is the first system to enable high-coverage, inexpensive error detection capability tailored to the stringent power/performance/area (PPA) constraints of HPC and data centers. With unobtrusive modification, the next generation of servers can achieve both strong comprehensive guarantees, and high-coverage sampling when uncompromised performance is the highest priority. The emerging blight of silent-error detection can be mitigated more effectively at the hardware level than with software scanners alone, and heterogeneity, achieved via induction parallelism, realises efficient guarantees otherwise only provided by expensive lockstep. Since ParaVerser repurposes compute units rather than adding dedicated, high-overhead components, systems that need guarantees and systems that do not will be able to make use of the same devices. ParaVerser offers, for the first time, the promise of a practical solution to silent data corruption at scale. Source code is available for this project at https://doi.org/10.5281/zenodo.15080017 or https://github.com/CompArchCam/DSN25-AE.

[20]To allow us to run these workloads to completion, and thus cancel out the effects of multicore workloads requiring more instructions for the same amount of work, the compute experiments were run on a Rockchip RK3588 SoC with A76 big cores at 2.4GHz, and A55 small cores at 1.8GHz.

REFERENCES

[1] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *CoRR*, vol. abs/2102.11245, 2021.

[2] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *HotOS*, 2021.

[3] K. T. Nguyen, "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, 2017.

[4] Arm Ltd., "Arm cortex-x2 core technical reference manual. cache protection behavior," 2023.

[5] I. B. Daniel Henderson, "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," tech. rep., IBM Systems Group, 2021.

[6] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," in *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[7] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *CASES*, 2003.

[8] D. F. Bacon, "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*, 2022.

[9] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," in *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[10] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The Arm triple core lock-step (TCLS) processor," *ACM Trans. Comput. Syst.*, vol. 36, June 2019.

[11] S. Ainsworth and T. M. Jones, "Parallel error detection using heterogeneous cores," in *DSN*, 2018.

[12] S. Ainsworth and T. M. Jones, "Paramedic: Heterogeneous parallel error correction," in *DSN*, 2019.

[13] S. Ainsworth, L. Zoubritzky, A. Mycroft, and T. M. Jones, "Paradox: Eliminating voltage margins via heterogeneous fault tolerance," in *HPCA*, 2021.

[14] R. Smith, "Arm announces Neoverse V2 and E2: The next generation of Arm server CPU cores." https://www.anandtech.com/show/17575/arm-announces-neoverse-v2-and-e2-the-next-generation-of-arm-server-cpu-cores, 2022.

[15] P. Alcorn, "Amd to make hybrid CPUs, also using AI for chip design: CTO Papermaster at ITF World." https://www.tomshardware.com/news/amd-to-make-hybrid-cpus-using-ai-for-chip-design-cto-papermaster-at-itf-world, 2023.

[16] J. Tyrrell, "Efficiency gains: predictive maintenance supports data center operations." https://techhq.com/2022/07/machine-learning-data-center-maintenance/, 2022.

[17] E. Maricau and G. Gielen, "Transistor aging-induced degradation of analog circuits: Impact analysis and design guidelines," in *ESSCIRC*, 2011.

[18] N. Karystinos, O. Chatzopoulos, G.-M. Fragkoulis, G. Papadimitriou, D. Gizopoulos, and S. Gurumurthi, "Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation," in *ISCA*, 2024.

[19] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *DATE*, 2011.

[20] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," in *DAC*, 2006.

[21] S. Hukerikar and N. Saxena, "Runtime fault diagnostics for GPU tensor cores," in *ITC*, 2022.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *CGO*, 2005.

[23] M. Didehban and A. Shrivastava, "NZDC: A compiler technique for near zero silent data corruption," in *DAC*, 2016.

[24] K. Mitropoulou, V. Porpodas, and T. M. Jones, "COMET: Communication-optimised multi-threaded error-detection technique," in *CASES*, 2016.

[25] L. Bautista Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications," in *PPoPP*, 2014.

[26] L. Bautista-Gomez and F. Cappello, "Detecting silent data corruption for extreme-scale MPI applications," in *EuroMPI*, 2015.

[27] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC*, 2012.

[28] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *CCGrid*, 2015.

[29] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for hpc applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, 2016.

[30] X. Ni and L. V. Kale, "Flipback: automatic targeted protection against silent data corruption," in *SC*, 2016.

[31] Z. He, Y. Huang, H. Xu, D. Tao, and G. Li, "Demystifying and mitigating cross-layer deficiencies of soft error protection in instruction duplication," in *SC*, 2023.

[32] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, "Mitigating silent data corruptions in hpc applications across multiple program inputs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022.

[33] D. Henderson, "Power8 processor-based systems RAS," tech. rep., IBM Systems Group, 2016.

[34] M. I. . Strategy, "AMD EPYC brings new RAS Capability," tech. rep., 2017.

[35] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, 2014.

[36] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *SC*, 2017.

[37] S. Di, H. Guo, E. Pershey, M. Snir, and F. Cappello, "Characterizing and understanding hpc job failures over the 2k-day life of ibm bluegene/q system," in *DSN*, 2019.

[38] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *DSN*, 2014.

[39] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *DSN*, 2015.

[40] G. Papadimitriou and D. Gizopoulos, "AVGI: Microarchitecture-driven, fast and accurate vulnerability assessment," in *HPCA*, 2023.

[41] O. Chatzopoulos, G. Papadimitriou, V. Karakostas, and D. Gizopoulos, "Gem5-marvel: Microarchitecture-level resilience analysis of heterogeneous soc architectures," in *HPCA*, 2024.

[42] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *ISCA*, 2021.

[43] T. J. Slegel, R. M. Averill III, 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, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, 1999.

[44] A. Wood, "Data integrity concepts, features, and technology," White paper, Tandem Division, Compaq Computer Corporation, 1999.

[45] N. Werdmuller, "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, 2021.

[46] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *DSN*, 2007.

[47] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO*, 2007.

[48] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *MICRO*, 2003.

[49] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "A self-tuning DVS processor using delay-error detection and correction," *IEEE Journal of Solid-State Circuits*, vol. 41, April 2006.

[50] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor II: In situ error detection and correction for PVT and SER tolerance," in *ISSCC*, 2008.

[51] S. Beer, M. Cannizzaro, J. Cortadella, R. Ginosar, and L. Lavagno, "Metastability in better-than-worst-case designs," in *ASYNC*, 2014.

[52] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems," in *MICRO*, 2009.

[53] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *ASPLOS*, 2008.

[54] P. Ramachandran, *Detecting and recovering from in-core hardware faults through software anomaly treatment*. PhD thesis, University of Illinois, 2011.

[55] P. Ramachandran, S. K. S. Hari, S. V. Adve, and H. Naeimi, "Understanding why symptom detectors work by studying data-only application values," in *Workshop on silicon errors in logic–system effects (SELSE)*, 2011.

[56] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *ASPLOS*, 2010.

[57] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: Building high availability systems with commodity multi-core processors," in *ISCA*, 2007.

[58] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults," in *PACT*, 2008.

[59] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ISCA*, 2009.

[60] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Necromancer: Enhancing system throughput by animating dead cores," in *ISCA*, 2010.

[61] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "The StageNet fabric for constructing resilient multicore systems," in *MICRO*, 2008.

[62] S. Gupta, A. Ansari, S. Feng, and S. Mahlke, "StageWeb: Interweaving pipeline stages into a wearout and variation tolerant CMP fabric," in *DSN*, 2010.

[63] A. Naithani, S. Eyerman, and L. Eeckhout, "Reliability-aware scheduling on heterogeneous multicore processors," in *HPCA*, 2017.

[64] T. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *MICRO*, 1999.

[65] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *ISCA*, 2002.

[66] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ISCA*, 2000.

[67] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *DSN*, 2007.

[68] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO*, 2007.

[69] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling + DMR: Practical and low-overhead permanent fault detection," in *ISCA*, 2011.

[70] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *RAMS*, 2008.

[71] A. Cunningham, "Arm goes 64-bit with new cortex-a53 and cortex-a57 designs." https://arstechnica.com/information-technology/2012/10/arm-goes-64-bit-with-new-cortex-a53-and-cortex-a57-designs/, 2012.

[72] J. Olšan, "big.LITTLE by AMD: Zen 4c has the same IPC as the big Zen 4." https://www.hwcooling.net/en/big-little-by-amd-zen-4c-has-the-same-ipc-as-the-big-zen-4-architecture-analysis/, 2023.

[73] Intel, "How 13th gen intel core processors work." https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html, 2022.

[74] Chaos Community, "Principles of chaos engineering." http://principlesofchaos.org/, 2019.

[75] T. Benacchio, L. Bonaventura, M. Altenbernd, C. D. Cantwell, P. D. Düben, M. Gillard, L. Giraud, D. Göddeke, E. Raffin, K. Teranishi, and N. Wedi, "Resilience and fault tolerance in high-performance computing for numerical weather and climate prediction," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, 2021.

[76] T. P. M. The next platform, "Fujitsu to fork arm server chip line to chase clouds." https://www.nextplatform.com/2023/03/15/fujitsu-to-fork-arm-server-chip-line-to-chase-clouds/, 2023.

[77] A. Ganti, "Exynos 2400, 2200, 2100 die shots highlight the chips' evolution over the years." https://www.notebookcheck.net/Exynos-2400-2200-2100-die-shots-highlight-the-chips-evolution-over-the-years.832760.0.html, 2024.

[78] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, "Understanding silent data corruptions in a large production cpu population," in *SOSP*, 2023.

[79] Intel, "OpenDCDiag." https://github.com/opendcdiag, 2025.

[80] J. Wittich, "How cpus will address the energy challenges of generative ai." https://www.infoworld.com/article/2336903/how-cpus-will-address-the-energy-challenges-of-generative-ai.html, 2024.

[81] https://www.arm.com/markets/artificial-intelligence/cpu-inference.

[82] A. Pellegrini, "Arm neoverse n2: Arm's 2nd generation high performance infrastructure cpus and system ips." https://hc33.hotchips.org/assets/program/conference/day1/20210818_Hotchips_NeoverseN2.pdf.

[83] "Arm cortex-x2 core technical reference manual r2p0, imp_cpuectlr_el1, cpu extended control register." https://developer.arm.com/documentation/101803/0200/AArch64-system-registers/AArch64-generic-system-control-register-summary/IMP-CPUECTLR-EL1--CPU-Extended-Control-Register.

[84] D. Schor, "Arm introduces the cortex-x4, its newest flagship performance core." https://fuse.wikichip.org/news/7531/arm-introduces-the-cortex-x4-its-newest-flagship-performance-core, 2023.

[85] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *MICRO*, 2019.

[86] Q. H. Dang, "Secure hash standard," tech. rep., NIST, 2015.

[87] M. W. Rashid and M. C. Huang, "Supporting highly-decoupled thread-level redundancy for parallel programs," in *HPCA*, 2008.

[88] J. Tchórzewski and A. Jakóbik, "Theoretical and experimental analysis of cryptographic hash functions," *Journal of Telecommunications and Information Technology*, 2019.

[89] B. Zhang, S. Ainsworth, L. Mukhanov, and T. M. Jones, "Parallaft: Runtime-based cpu fault tolerance via heterogeneous parallelism," in *CGO*, 2025.

[90] Arm Ltd., "Arm cortex-x2 core software optimization guide." https://developer.arm.com/documentation/PJDOC-466751330-14955/latest, 2021.

[91] WikiChip, "Cortex-X1 - Microarchitectures - ARM." https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-x1, 2023.

[92] A. Frumusanu, "Arm announces mobile Armv9 cpu microarchitectures: Cortex-X2, Cortex-A710 & Cortex-A510." https://www.anandtech.com/show/16693/arm-announces-mobile-armv9-cpu-microarchitectures-cortexx2-cortexa710-cortexa510, 2021.

[93] Arm Ltd., "Arm cortex-a510 core software optimization guide." https://developer.arm.com/documentation/PJDOC-466751330-536816/latest, 2021.

[94] WikiChip, "Cortex-A510 - Microarchitectures - ARM." https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a510, 2023.

[95] Arm Ltd., "Arm cortex-a55 software optimization guide." https://developer.arm.com/documentation/EPM128372/0400/?lang=en, 2022.

[96] A. Frumusanu, "Arm announces new Cortex-A35 CPU - ultra-high efficiency for wearables & more." https://www.anandtech.com/show/9769/arm-announces-cortex-a35, 2015.

[97] Arm Ltd., "Arm Cortex-A processor comparison table." https://developer.arm.com/documentation/102826/0500, 2024.

[98] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[99] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[100] A. Frumusanu, "Arm announces Neoverse V1, N2 Platforms & CPUs, CMN-700 mesh: More performance, more cores, more flexibility." https://www.anandtech.com/show/16640/arm-announces-neoverse-v1-n2-platforms-cpus-cmn700-mesh/7, 2021.

[101] D. Schor, "Tsmc n3, and challenges ahead." https://fuse.wikichip.org/news/7375/tsmc-n3-and-challenges-ahead/, 2023.

[102] D. Schor, "Samsung 3nm gaafet enters risk production; discusses next-gen improvements." https://fuse.wikichip.org/news/6932/samsung-

3nm-gaafet-enters-risk-production-discusses-next-gen-improvements/, 2022.
[103] 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 *MICRO*, 2009.