Parallel Error Detection Using Heterogeneous Cores

Sam Ainsworth, Timothy M. Jones
University of Cambridge, UK
{sam.ainsworth,timothy.jones}@cl.cam.ac.uk

***Abstract*—Microprocessor error detection is increasingly important, as the number of transistors in modern systems heightens their vulnerability. In addition, many modern workloads in domains such as the automotive and health industries are increasingly error intolerant, due to strict safety standards. However, current detection techniques require duplication of all hardware structures, causing a considerable increase in power consumption and chip area. Solutions in the literature involve running the code multiple times on the same hardware, which reduces performance significantly and cannot capture all errors.**

We have designed a novel hardware-only solution for error detection, that exploits parallelism in checking code which may not exist in the original execution. We pair a high-performance out-of-order core with a set of small low-power cores, each of which checks a portion of the out-of-order core's execution. Our system enables the detection of both hard and soft errors, with low area, power and performance overheads.

***Keywords*—fault tolerance; microarchitecture; error detection**

I. Introduction

Hardware faults, both soft (transient) and hard (permanent), are increasingly common in microprocessors. As technology nodes reduce and the number of transistors in a system increases, the likelihood of a failure is heightened. Small transistors are more vulnerable to transient errors caused by cosmic rays, and increased variability at smaller feature sizes significantly increases the occurrence of transient faults [1].

At the same time, the tolerance of many workloads to the occurrence of errors has reduced. For example, strict safety standards, along with suboptimal environmental conditions, require error detection hardware within CPUs used for automotive, health, nuclear power and machinery applications [2], [3], [4]. Space applications require reliability for economic reasons [3], and large scale HPC systems require reliability due to having a large number of potential failures [5], [6], [7].

Certain industries mandate stringent safety standards to achieve certification, such as automotive, where redundancy is required for ASIL-C and ASIL-D ratings [8]. The current industry approach to address this is hardware lock-step error detection [3], [9], [10]. This involves running multiple copies of a program on separate, synchronised CPUs, and comparing the results in hardware. However, this is both energy- and silicon-area-intensive. As the computational requirements of these systems grows [2], out-of-order cores are rapidly becoming necessary to achieve the required performance. Duplicating out-of-order cores comes at too high cost in energy, heat dissipation and area to be practical.

Redundant multi-threading [1], [11], [12] has been proposed, where simultaneous threads on the same core are used to run two copies of a program, and the results compared. However, as the same hardware is used for both copies, permanent faults within the core can only be detected though the addition of extra logic [13], and performance is also significantly reduced over the same code without error detection.

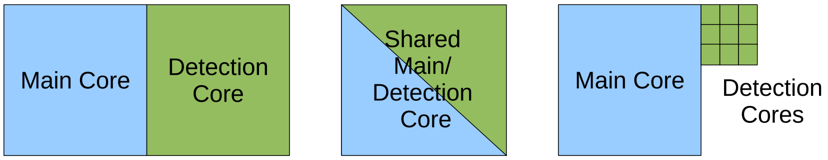
Prior work [14] has noted that parallelism is available in the second, error-detecting run of a computation, and thus it is possible to use homogeneous multi-core processors to reduce energy cost by using dynamic frequency-voltage scaling at the expense of tripling silicon area. In our approach, we focus on a heterogeneous architecture specialised for exploiting far larger amounts of error detection parallelism. We use the principle of strong induction [15] to dramatically reduce the overheads of using hardware to detect errors. We perform delayed error detection on multiple small cores [16], which check the computation carried out on a high-performance out-of-order core. By taking periodic register checkpoints and tracking the loads and stores carried out on the main core, tiny checker cores can independently verify a portion of the original computation each. The heterogeneity between the computation core and its coupled checker cores allows us to reduce area and power overheads significantly, while maintaining high performance.

II. Background

We discuss the increasing prevalence of faults in microprocessors and standard error-detection schemes, before motivating our approach in section III.

A. Faults

Hardware faults fall into one of two distinct categories. Permanent faults are the result of errors during manufacture or wearout during the service life of the system. Transient faults, on the other hand, are caused by strikes from cosmic rays and alpha particles, and do not persist. However, there is no effective way to shield a microprocessor from cosmic rays [1], and while smaller transistors are individually less likely to be hit by a ray, increasing numbers of transistors in modern systems, coupled with the nominal energy required to switch a transistor at low source voltages [17], makes chips increasingly vulnerable to transient faults [18]. Rapidly increasing core counts for workloads such as HPC mean there are more points of failure in systems, and therefore a higher chance of hard faults [5], [6], [7]. Increased variance in chips at lower source voltages [17], [19] make timing violations more common, and the unfavorable conditions many safety critical systems operate in, such as those in space or the automotive industry [2], [3], [9], also serve to increase the number of faults observed in modern systems.



(a) Lockstep (b) RMT (c) Desired

Overhead	Lockstep	RMT	Desired
Area	Large	Small	Small
Energy	Large	Large	Small
Performance	Negligible	Large	Negligible

(d) Comparison

Fig. 1: Running cores in lockstep, or running the same code twice on the same core via multi-threading, come with significant space and time overheads, respectively. However, we can optimise all of these significantly if we can exploit parallelism in the detection to run it on separate, simpler processors.

B. Detection

Current detection schemes use a combination of space and information redundancy to cover faults [1]. Information redundancy refers to using error-detecting codes, such as ECC, to detect and correct errors in stored and transmitted data. As errors in main memory are common, systems requiring reliability, such as servers, have long been covered by ECC [20].

However, information redundancy techniques do not extend to error checking within the computation itself, and thus the processor logic must be covered by other schemes. Current reliable systems favor lock-step error detection [3], [9], [10], a space-based redundancy scheme where cores are duplicated in their entirety. The program is executed on both simultaneously, perhaps with some delay on the second core to avoid correlated transient errors, and the results compared. This comes at both a high chip area and power cost. Some techniques, such as DIVA [21], [22], attempt to get around this by using simplified duplicate hardware at the end of the pipeline, removing some of the repeated work at the expense of requiring ECC on all architectural state, including the register file. While some architectures have featured parity bits on such state [23], full ECC is too invasive to the microarchitecture and hence such techniques have not seen implementation. Many time-based schemes [1], [11], [12], [24], [25], which run duplicates of a program on the same core twice at different times and compare the results, have been proposed, exploiting techniques such as simultaneous multi-threading to improve performance. However, their high performance penalty and inability to cover hard faults with additional logic [13] have meant that these schemes have seen little use in practice.

III. Motivation

The performance demands of common workloads are ever increasing. As processors tend to be energy or heat limited, due to current trends in silicon scaling [19], this translates into a demand for high performance at low power. Hardware duplication comes at the expense of performance, by reducing the power budget. Indeed, the cost of duplicating high-performance out-of-order superscalar systems is typically too high for practical error detection, as out-of-order superscalar systems are already inefficient [26].

We require fault detection with high performance, low power consumption and low chip area. Figure 1 shows how existing techniques accomplish these. Lockstepping [9] (fig. 1(a)), where we run the same program on two identical processors, only provides the former. Redundant multi-threading [1], [11], [12] (fig. 1(b)) only provides the latter.

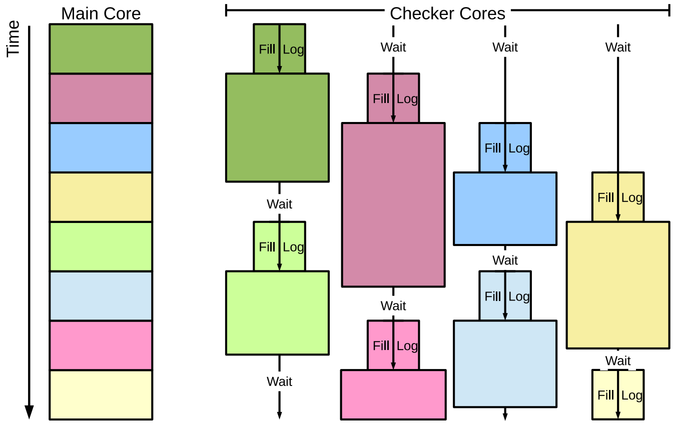


Fig. 2: We use register checkpoints to split dynamic execution from a main core into small instruction streams. These are run again on one of several small checker cores, to verify execution. Each stream is independent of all others, allowing parallelization of error detection.

If we can exploit heterogeneity between computation and detection (fig. 1(c)), we can achieve the low overheads and minimal design invasiveness we require. In general, the smaller the CPU, the more useful work can be done per unit area and power [27]. For example, as of November 2017, the Green500 list [28] is topped by machines using arrays of tiny cores. Modern GPUs and the Xeon Phi [29] follow a similar design philosophy. Therefore, one way of achieving our goals, provided we can parallelise the checking code, is to run the fault detection on a set of very low power cores. However, common CPU workloads tend to exhibit very little thread level parallelism, so large out-of-order cores are necessary to attain high performance [29], [30]. Despite this limitation, in the next section we develop a scheme which realises the parallelization of error detection for any program, enabling us to take advantage of heterogeneity between the main and checker cores, achieving full error detection with only minor area, power and performance overheads.

IV. Parallel Error Detection

Our approach to error detection parallelizes the execution of fault checking code, even when the original program is sequential. We use the principle of strong induction [15] to check multiple parts of the executed program at once. In other words, we check each part of the application independently assuming all previous parts were correct. Provided we prove this for each part of the program, it is possible to ensure the entire workload is free of hardware faults.

To realise parallel fault detection, we repeat computation from a main high performance out-of-order core by executing duplicate copies of all instructions. We take periodic register checkpoints and use these to spawn checker threads which repeat all computation between two checkpoints, executing asynchronously and in parallel, since they are independent of each other. During the original execution, we log load and store values and addresses, redirecting duplicated memory accesses from checker threads to the log, which allows the main application to overwrite memory locations without restriction. The checker threads, executing on multiple low-power cores, read the same memory values as the main core did, and check the addresses and values of stores, and addresses of loads. Figure 2 gives an example of how execution proceeds.

Each checker thread assumes that its starting checkpoint is correct. It repeats all instructions up to its end register checkpoint, which it validates, and which is also the starting checkpoint for another checker thread. Additional hardware checks stored data and their addresses against those from the original computation as the checker threads execute.

We allow values potentially affected by a fault to propagate into main memory. This is necessary to both avoid slowing down the main core (as would occur with a large forwarding table to forward unchecked stores), and to allow a large number of loads and stores be checked simultaneously, yielding the parallelism that we exploit. This is common in software schemes [31], [32]. If a check fails (either a check on a store or a register checkpoint validation), all future computation must be assumed to be faulty. This is because the assumption of correctness of previous computation, required for the strong induction hypothesis, does not hold. Correctness is only known once all checks up to a given point successfully complete. Similarly, if an error is detected within a check, we do not know if it was the first error until all previous checks complete. Once that happens, our system provides sufficient information to identify that a fault has occurred, and the position of that first error, giving a practical error detection mechanism.

A. Overview

Figure 3 gives an overview of our system. We attach a collection of small checker cores to a conventional out-of-order core, in order to efficiently execute the duplicated instructions. The loads and stores performed by the main core are stored in a hardware load-store log [1], [11], [12], which is then split into multiple segments, each checked by a different checker core in parallel. There is a one-to-one mapping between log segments and checker cores. The checker cores are also given a copy of the register file at the start and end of each segment, from which to start execution.

Loads are duplicated early by the load forwarding unit, to ensure any errors within loaded values in the main CPU don't propagate to the checker cores. We assume memory blocks such as caches and DRAM are protected by ECC, since our detection scheme is only designed to cover errors within the core. Loaded values are copied within the cache at a point still protected by ECC, ensuring that errors from the main core's load cannot propagate to the checker cores. We further assume the instruction stream is read-only, such that the instructions read by checker units will be identical to those read by the main thread. This is a common design choice [1], [12], where modifications to the instruction stream (e.g., in the case of self-modifying code) require all checking to complete first.

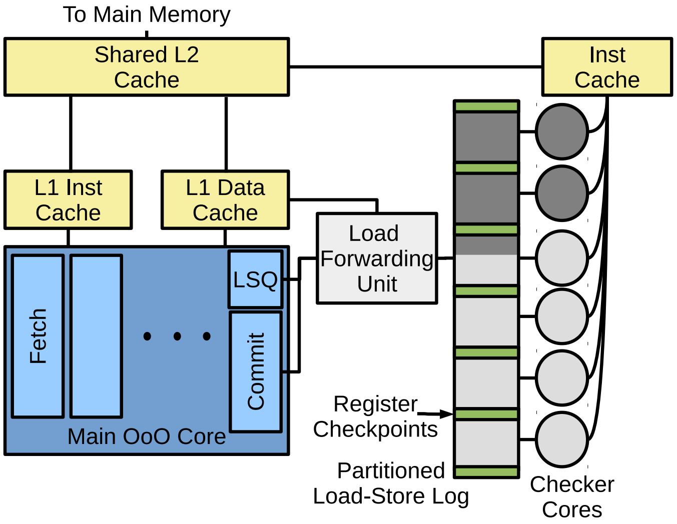


Fig. 3: Error detection is performed in parallel on a set of small checker cores that execute duplicate instructions between two register checkpoints, reading memory values from a load-store log and validating store addresses and data.

Our scheme provides only detection, rather than correction, of soft and hard errors. This is equivalent to the dual-core lockstep techniques used typically in the automotive industry [33], where detection is mandatory and correction unnecessary, and thus only detection is typically provided [9]. The detection of an error triggers an exception within a program, which can either be caught and handled, or cause termination of the application, as with dual core lockstep implementations [34]. Errors detected within kernel code are reported to the kernel itself. Incorrect values are deliberately allowed to propagate into main memory and devices on a detected error: the exception trigger's semantics take this into account.

Ours is a pure hardware scheme where detection is performed without original program modification. The main core and checker cores execute identical code: differing load and store behavior on checking, and stopping on reaching a register checkpoint, are achieved using hardware logic. This is similar to many redundant multi-threading schemes [1], [11], [12].

B. Checker Cores

Each of our small checker cores must implement the same ISA as the main core, so that all cores can execute the same

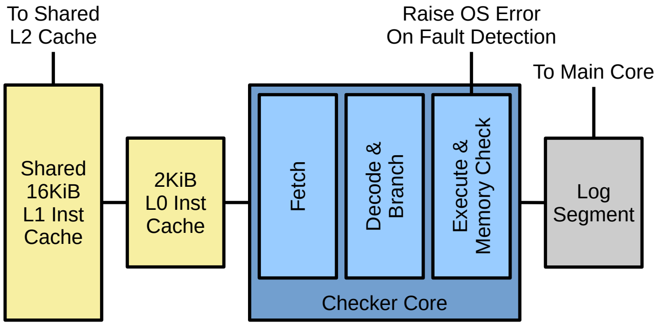
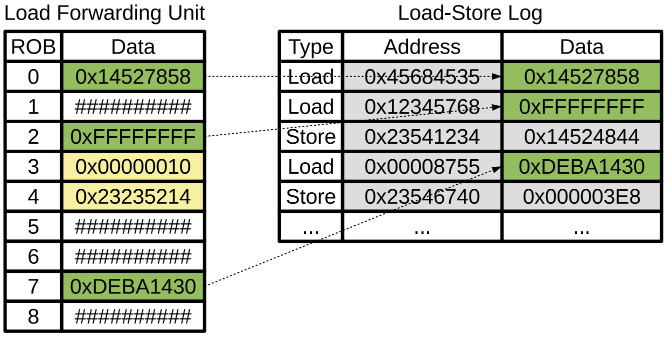


Fig. 4: The in-order checker cores have a short pipeline, private L0 instruction cache and L1 instruction cache shared between all checker cores. They read data from their load-store log segment and validate store addresses and data.


Load Forwarding Unit

ROB	Data
0	0x14527858
1	##########
2	0xFFFFFFFF
3	0x00000010
4	0x23235214
5	##########
6	##########
7	0xDEBA1430
8	##########

Load-Store Log

Type	Address	Data
Load	0x45684535	0x14527858
Load	0x12345768	0xFFFFFFFF
Store	0x23541234	0x14524844
Load	0x00008755	0xDEBA1430
Store	0x23546740	0x000003E8
...	...	...

Fig. 5: Speculative load values are placed in the load forwarding unit. On commit, these values are forwarded to the load-store log, which holds non-speculative loads and stores.

instruction stream. However, as only architectural state needs to be checked for correctness, micro-architectural implementation specifics may differ. We take advantage of this to keep power and chip area overheads to a minimum, by using smaller checker cores than the main core. These are in-order, very small, and run at a low clock speed, meaning that we need several checker cores to keep up with the performance of the main core (in our experiments, we use 12). An example is shown in fig. 4.

The checker cores perform the same work as the main core, so many of the instructions executed are likely to be in the L2 cache. They are also likely to share code with each other. These factors, along with a limited area budget for instruction caches, lead to an L1 instruction cache shared between the checker cores, connected to the main core's L2, along with a set of very small L0 instruction caches for each checker core. The checker cores only access data from the log, rather than main memory, and all accesses to this structure are sequential, so a data cache is unnecessary.

A checker core starts once architectural register checkpoints are available for the start and end of its computation; the stream of loads and stores executed between will have been captured in the corresponding load-store log segment (as discussed in section IV-D). The checker core begins with the PC from the starting checkpoint. It executes the original instruction stream, but reads load values by looking up the next value in the log segment, and checking in hardware that the addresses match, instead of accessing a cache or memory. On a store, hardware logic checks both the address and stored value to ensure they are the same as in the log. If a check fails, an error exception is raised for the main core.

A checker core stops execution when the stream ends, as a result of reaching the last of the loads and stores for a segment (see section IV-D), or reaching a timeout instruction count (section IV-J). Following this, the register file is checked for consistency with the checkpoint taken at the end of the original stream, and then the checker core sleeps until another stream is ready to be checked.

C. Load Forwarding Unit

The main core and checker cores read the same memory addresses. However, the checker cores' executions lag behind the main core. This means that by the time the checker cores read the values in memory, they may differ from those that the main core read, resulting in incorrect execution. We therefore forward the results of loads from the main core into an SRAM log, for the checker cores to read.

If an error occurs after forwarding, it will be detected by the checking cores, provided it causes any stores, addresses, or the register file at the end of each checkpoint to differ (all other errors do not change state, and thus do not need to be detected). However, naïvely forwarding loaded values direct from the main core to the log introduces a window of vulnerability. If an error occurs to a loaded value in a physical register in the main core before the value is forwarded, the error will be duplicated in the checker core.

Our solution is to add a load forwarding unit. Loads from the cache are duplicated immediately and stored in this table, then forwarded to the load-store log at commit. This prevents any errors from the main core's loads propagating into the checker cores. Since there are always two copies of loaded values, errors within the loaded data in the main core don't get duplicated. As speculative loads can go into this table, each load is tagged with the associated reorder buffer ID assigned to the instruction. This is then used to select the actual loads that need to be forwarded at commit.

Similarly, loads forwarded from the core's load-store queue instead of from the cache are also sent to the load forwarding unit. This is sufficient to ensure full error detection, as any errors in the forwarded value will also propagate to the data stored to memory, and thus the value for the associated store in the log, and the check of the stored value on the second core will catch these errors.

We show this behavior in fig. 5. Speculative loads are added into the load forwarding unit from the cache. On a commit of a load instruction, the loaded value within the load forwarding unit, and the address, are output into the load-store log, shown in green. Mis-speculated loads, in yellow, and reorder buffer

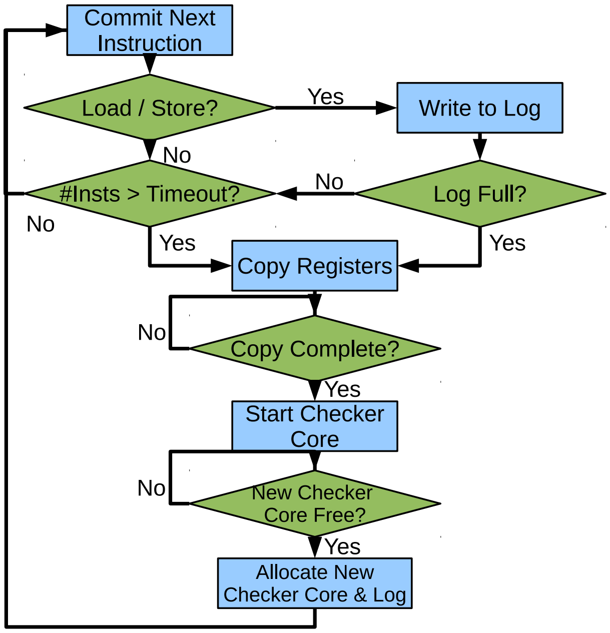


Fig. 6: A flow diagram detailing the interaction between the main core's commit stage and the load-store log.

entries containing non-loads, in white, do not get forwarded.

Having a load forwarding unit as large as the reorder buffer is over-provisioning because not all of the instructions going through a pipeline will be loads. Therefore, the table will never be full. However, by associating entries with reorder buffer IDs, we avoid having to flush incorrectly speculated loads from the load forwarding unit since they will be overwritten when the reorder buffer entries are reallocated. More advanced schemes could optimise the size of this table, but these are orthogonal to our work.

D. Partitioned Load-Store Log

We use an SRAM log structure to forward both the load data, for computation repetition, and the addresses and values of stores, to be checked against those computed by the checker cores for error detection. This information is collected in hardware, when the loads and stores on the out-of-order core commit. In this way, data is stored in the order it will be used on the in-order checker cores. Therefore, to forward a load or check a store, the next entry in the log simply needs to be read. The results of other non-deterministic instructions are forwarded in a similar way. The interaction between the main core and load-store log is shown in fig. 6.

Where our scheme differs from previous implementations of such a log [1], [12] is that ours is partitioned. This means that different parts of the log can be checked simultaneously by multiple checker cores. We achieve this by storing architectural register checkpoints from the main core whenever a segment of the load-store log is filled. We then start a checker core with the register checkpoint collected when the previous segment was filled. When a check completes, the relevant segment of the log is freed to be used again. If all log segments are full, we stall the main core until a checker core finishes and clears its queue. In practice, this is rare.

To detect all errors, while still only forwarding architectural state, we must start each checker core at the beginning of an architectural instruction, which may be a macro-op (an instruction that is split by the decoder into smaller, less complex operations, called micro-ops). If a micro-op from a partially executed macro-op fills the load-store log, we must copy all loads and stores caused by the currently executing macro-op into the next load-store log entry. An alternative solution would be to start filling a new log segment whenever there are fewer free entries in the current segment than required for the largest possible macro-op.

As shown in fig. 3, there is a one-to-one correspondence between checker cores and load-store log partitions. This simplifies data paths, so that no arbitration is required between logs and cores. However, it also means that either one of the checker cores or the main core must always be stalled (checker cores stall when their log segment is being filled, the main core stalls when there are no free log segments to write to). As each checker core is very small, it is preferable to include the extra core over having a complicated indirection layer to provide additional log segments, which would increase wiring.

E. Detection Trade-Offs

Although we divide the load-store log into multiple segments to attain checking parallelism, there is an inherent trade-off between the overheads in creating a segment and the latency of error detection. Each time we fill a segment, an architectural register checkpoint must be taken within the main core, which involves copying a large set of registers. To make this cost negligible, we need to reduce the frequency at which it occurs, which is achieved by increasing the size of each segment. However, as segments grow larger, the time taken to fill each one increases, as does the time taken for a checker core to check it, therefore increasing the average latency between an error and its detection.

Our scheme provides two methods to adjust this trade-off. One the one hand, we can vary the number of segments while maintaining the same total size of the log. This affects the amount of parallelism available, since it results in a corresponding change in the number of checker cores. Lower degrees of parallelism mean the checker cores must be more aggressive, or clocked at a higher frequency, to enable error detection to keep up with the performance of the main core. On the other hand, we can vary the size of the load-store log, such that each segment is larger, which has obvious implications for the on-chip storage requirements. We initially choose values that favor low overheads for the main core with manageable detection latencies, then further explore these costs in section VI-A.

F. Memory System

Parallel error detection inevitably results in increased latency between the original execution and checking of a given instruction, compared with a lock-step scheme, which is necessary to achieve parallelism. This means that holding back stores until they have been checked is unappealing: adding indirection for load-store forwarding for this number of stores would slow down the common case of error-free execution.

In our approach, we therefore allow potentially-faulty stores to escape into memory, as is common with software error detection schemes [31], [32], and let error handling software deal with correction if necessary. Suitable correction techniques for these circumstances, if required, include checkpointing [35], write-ahead logging [36] and transactions [37], both in hardware and software. However, in many applications, such as in the automotive sector, rather than correcting the software, the system is likely to be restarted [34], [38], and thus rollback correction is unnecessary.

G. Interrupts

For the stream of (committed) loads and stores seen by the main core and checker cores to be identical, the checker cores must see interrupts at the same point in the code as the main core. To address this, we finish segments based on interrupts by issuing an early register checkpoint on the interrupt boundary. This also occurs when the processor context switches, to provide easier fault reporting. In this case a new checkpoint is created, the check for which continues running after the context switch, and data from the new context is placed in a new log entry.

Although this may slightly reduce the occupancy of the load-store log segments, this is negligible due to the infrequent nature of interrupts. Another solution is to insert interrupt events into the load-store log when they reach the commit stage of the main core's pipeline. Although a good choice in certain designs [1], this involves greater modifications to the main core for our scheme.

H. System Faults

Our error detection scheme assumes that errors are reported to the program itself. However, some errors cause early termination of an application before they are checked, such as segmentation faults. To avoid this, we hold back the termination of processes until the checker cores have finished execution. If the check succeeds, we terminate the program. Otherwise, the operating system issues a fault-detection error, to be dealt with by the application, with a default handler terminating the process.

I. Over-Detection

The addition of redundant logic causes more errors to occur within a system by necessity, because more components exist, each of which can introduce new faults. Errors within the checker circuitry do not affect the main program. However, on detecting a fault, we cannot verify which of the main core and checker core produced the incorrect result, so we report all errors to the operating system. Since the additional area requirements of our technique are small (see section VI-B) errors in detection components are less common than those in the main core and so false-positives are rare.

For our system to catch all errors, we need to check all stores, the addresses of all loads, and also the register checkpoints at the end of each log segment. Previous work [1] has established that only stores and load addresses need to be checked for correctness, as register state is never visible outside of the processor. However, the ability to check from multiple locations in parallel relies on an induction hypothesis: each individual check verifies that loads and stores are correct, assuming the register file and previous loads and stores were correct up to that point. By checking the register file at the end of each checkpoint, we can combine each individual check to cover the whole program.

However, this adds an additional over-detection source. Registers which are checked for errors may not impact any future loads or stores because they may be overwritten without being used. Since register liveness is only made evident in future partition checks, it is not possible to calculate whether this is the case. We must therefore report an error even if it may not cause problems in future segments. Increasing the size of each load-store log segment reduces the already negligible false-positive rate from this, but increases detection latency and storage requirements.

J. Timeouts

The primary means of starting a check on an instruction stream is the filling of a load-store log segment. Likewise, an instruction stream is considered error-free once its corresponding log segment and final register checkpoint have been validated. While this maximises the utilization of the fixed-sized load-store log, there are cases when we may wish to trigger detection early.

For example, the main core could erroneously enter an infinite control-flow loop with no loads or stores, meaning the log segment would never be filled and no new checks would be issued. Similarly, the checker core may do the same upon an error affecting it, meaning the check would never complete and the error never be detected (even though this scenario corresponds to over-detection, see section IV-I).

To solve this, we introduce a timeout value, which corresponds to a maximum number of instructions in the stream for each log segment. A check is therefore started on a checker core when either the main core fills a load-store log segment or it reaches this maximum instruction count. Figure 6 shows this interaction. We then validate the register checkpoint either when all loads and stores have been checked in the load-store log segment, or when the number of committed instructions is equal to the number committed on the original core. This maximum instruction count simultaneously solves the issue of either type of core getting stuck in an infinite loop. For the main core it means we must always eventually attempt to validate the most recent stream of instructions. For the checker cores, if we reach our maximum number of instructions without having checked all loads and stores in the log segment, we know that execution has diverged.

Termination before the load-store log segment is filled is also useful even under correct execution. By allowing early

Main Core	
Core	3-Wide, out-of-order, 3.2GHz
Pipeline	40-Entry ROB, 32-entry IQ, 16-entry LQ, 16-entry SQ, 128 Int / 128 FP registers, 3 Int ALUs, 2 FP ALUs, 1 Mult/Div ALU
Tournament	2048-Entry local, 8192-entry global, 2048-entry
Branch Pred.	chooser, 2048-entry BTB, 16-entry RAS
Reg. Checkpoint	16 cycles latency
Memory	
L1 ICache	32KiB, 2-way, 2-cycle hit lat, 6 MSHRs
L1 DCache	32KiB, 2-way, 2-cycle hit lat, 6 MSHRs
L2 Cache	1MiB, 16-way, 12-cycle hit lat, 16 MSHRs, stride prefetcher
Memory	DDR3-1600 11-11-11-28 800MHz
Checker Cores	
Cores	12× In-order, 4 stage pipeline, 1GHz
Log Size	36KiB: 3KiB per core, 5,000 instruction timeout
Cache	2KiB L0 ICache per core, 16KiB shared L1

TABLE I: Core and memory experimental setup.

Benchmark	*Source*	*Input*
randacc	HPCC [39]	100000000
stream	HPCC [39]	
bitcount	MiBench [40]	75000
blackscholes	Parsec [41]	simsmall
fluidanimate	Parsec [41]	simsmall
swaptions	Parsec [41]	simsmall
freqmine	Parsec [41]	simsmall
bodytrack	Parsec [41]	simsmall
facesim	Parsec [41]	simsmall

TABLE II: Summary of the benchmarks evaluated.

detection triggering, we can split streams based on hardware events such as interrupts, for example (see section IV-G), simplifying event ordering for the checker cores.

K. Summary

We have discussed the hardware required to parallelise error detection to a set of small cores. These cores observe the loads and stores committed by the main core and use them to replay the instructions executed by the main core. Loads and stores are split up within a partitioned load-store log, and separated by register checkpoints, allowing each checker core to work on a different part of the main core's execution simultaneously.

A checker core starts execution after either its segment of the load-store log is filled, or a timeout value is reached. On error detection, the fault is reported to the program, which must then either terminate execution or return the memory system to a consistent state from which execution can restart.

Our scheme allows error detection with minimal power, performance and area overheads, by trading off detection latency for parallelism. The next sections quantify how each of these are affected by our scheme.

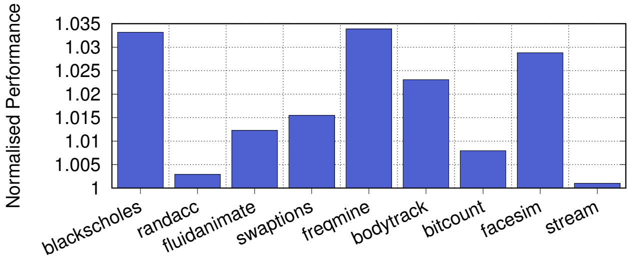


Fig. 7: Normalised slowdown for each benchmark, at standard settings (table I).

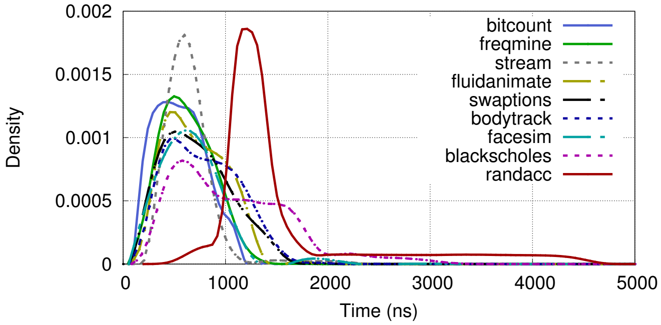


Fig. 8: Density plot, to show the distribution of error detection delays, at standard settings (table I).

V. Experimental Setup

To evaluate the checker core performance required for our error detection technique, along with the latency between an error and its detection, we modeled a high performance system using the gem5 simulator [42] with the ARMv8 64-bit instruction set and configuration given in table I, similar to systems validated in previous work [43]. A summary of the benchmarks we evaluated is given in table II. We used benchmarks taken mostly from Parsec [41], as a modern benchmark suite representative of a wide range of workloads. In addition, we chose RandomAccess and STREAM from the HPCC benchmark suite [39] and Bitcount from MiBench [40] to evaluate applications at the extremes of being almost purely memory bound (both irregular and regular) and almost purely compute bound, respectively. We choose these benchmarks to give both a wide-ranging suite of applications, along with extreme and worst-case behaviour, to analyze the entire range of performance overheads.

VI. Evaluation

Figure 7 shows the performance impact of our parallel error detection with the checker cores running at default settings, as given in table I. The average slowdown is 1.75%, and no benchmark slows down by more than 3.4%. Performance overheads are primarily caused by the time taken to checkpoint registers at the end of a load-store log segment.

We plot the distribution of delays between loads and stores being executed and checked, for each benchmark, in fig. 8. Each resembles a normal distribution, with the benchmarks featuring more homogeneous workloads (randacc, stream,

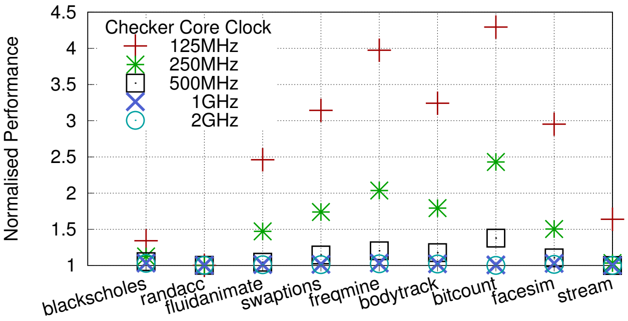


Fig. 9: Normalised slowdown when varying the frequency of the checker cores.

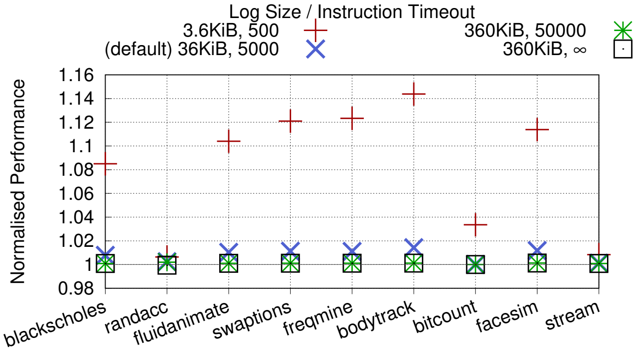


Fig. 10: Slowdown to the system from just checkpointing, without any checker core execution, across queue sizes and instruction timeouts.

facesim) most closely matching this. The highest average delay, of 1550ns, comes from randacc. This application is highly memory bound, with little temporal or spatial locality, resulting in a low IPC. This means that each load-store log segment takes a long time to fill, and the 5,000 instruction window timeout is lengthy compared to other benchmarks.

Each distribution features a long, but very thin, tail at the far right of the distribution: the maximum detection delay is significantly higher for every benchmark, at an average of 21.5μs. These points are not shown on the distribution plot, as they are too uncommon: for all benchmarks, 5000ns is sufficient to cover over 99.9% of all loads and stores.

For automotive applications, the faults we wish to avoid are based on physical motions. These occur on the timescale of milliseconds to seconds, so both the maximum and mean delays introduced by our scheme are acceptable. Similarly, for HPC workloads, checkpoints are performed at a frequency of no more than several minutes [5], [44], so delays introduced by our scheme are insignificant. As sections VI-B and VI-C show, overheads compared with dual-core lockstep, which we intend to replace, are greatly reduced.

A. Parameter Sensitivity

Our default configuration of the checker cores and load-store log prevents the majority of slowdowns, reflects a sensible trade-off in terms of performance and delay, and enables performance scaling across a number of checker cores. We evaluate these claims in the following sections.

Clock Frequency Figure 9 shows the performance impact of our scheme when varying the clock speed of the checker cores, compared to the default 1GHz. Since there are no data cache misses for the checker cores, because all loads and stores are accessed and checked from the load-store log, benchmarks which are memory bound, such as randacc and stream, do not experience significant performance losses, even at very low frequencies. However, others that are more compute bound, for example swaptions and bitcount, slow down significantly, particularly at clock speeds lower than 500MHz, because the checker cores combined do not have enough compute power to keep up with the main core. In this situation, the main core spends a significant amount of time stalled and waiting for load-store log space.

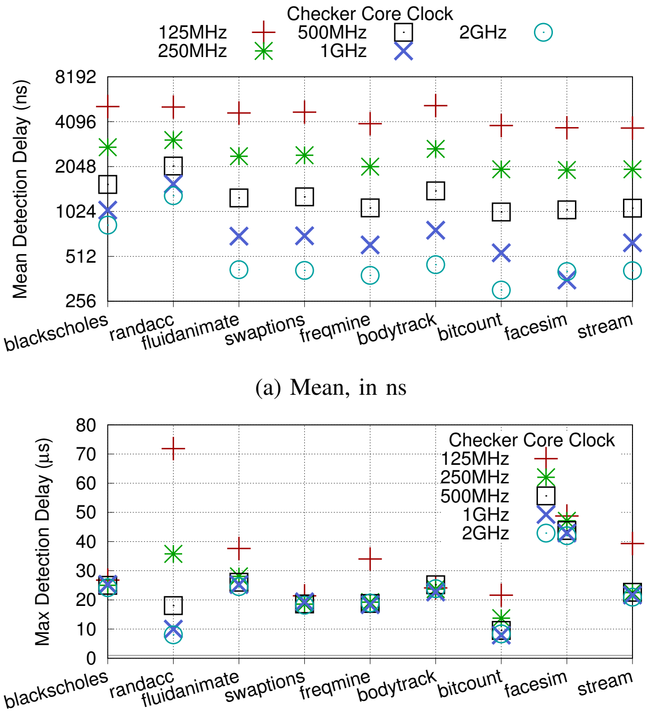
Register Checkpoint Overhead Even without the main core stalling from waiting for a free load-store log segment, our scheme still incurs some performance overhead from register checkpoint latency at the end of a segment. We assume a 16 cycle pause in commit when this occurs, allowing two-ported register files to copy 32 registers from each file.

The frequency of checkpointing is determined by the size of each load-store log segment, the instruction timeout, and delay properties of the system. Figure 10 shows the slowdown caused just by the checkpointing system, with increasing queue sizes and timeout lengths. The default 36KiB log is large enough to restrict slowdowns to no more than 2% across each of our benchmarks, with randacc being least affected due to its low IPC and thus infrequent checkpointing. A larger log, ten times the size, either with an associated ten times larger timeout or with an infinite timeout, is enough to reduce overheads to negligible amounts. By comparison, a ten times smaller log size and timeout length causes significantly higher overheads in most cases, with slowdowns of up to 15%.

We next discuss how these settings affect the observed delay. However, fig. 10 shows that a 36KiB log represents a good trade-off between silicon area and performance overhead.

Frequency Impact on Error Detection Delay As the goal of this technique is to scale error checking onto multiple cores, through parallelisation, inevitably the error detection time is higher than with would be with lockstep schemes, where errors are typically detected within a few cycles [3]. One method of controlling this delay is through the frequency of the checker cores, which is explored in fig. 11 for default load-store log sizes and timeout lengths. It shows mean and maximum delay between stores committing and being checked when varying the frequency of the checker cores.

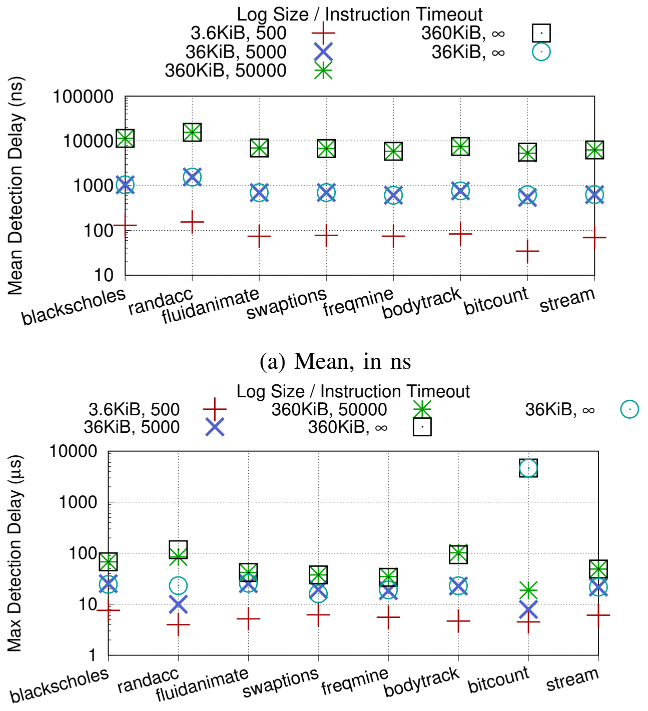
The mean detection delay is affected linearly by clock speed, in that doubling the clock speed approximately halves the delay. The exception to this is with high clock frequencies, where eventually the limiting factor becomes the time to fill the load store queue using the main CPU, rather than checking time. Maximum times are affected with less of a deterministic pattern. Maximum times are typically dictated by, for example,



(a) Mean, in ns

(b) Maximum, in μs

Fig. 11: Delay between a store committing and being checked, when varying the frequency of the checker cores.



(a) Mean, in ns

(b) Maximum, in μs

Fig. 12: Delay between a store committing and being checked, when varying the load-store log size and instruction timeout.

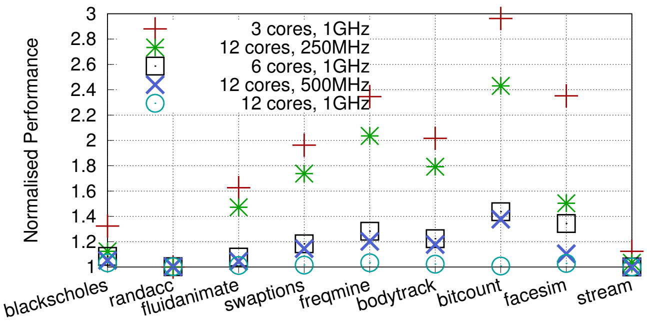


Fig. 13: Slowdown with varying core counts at 1GHz, compared with values for 12 cores at varying frequencies.

large numbers of cache misses on the main core, so altering the checker core frequency often does not affect these to such a significant extent.

Log Size Impact on Error Detection Delay Figure 12 shows mean and maximum detection delays when varying the load-store log size and timeout length, at the default checker core frequency. Mean detection times scale linearly with the load-store log size: a tenfold increase in log size and timeout results in a tenfold delay increase. While maximum detection times follow a similar trend, the pattern is more sporadic, due to individual instructions dominating the measurements.

For smaller log sizes and timeouts, many segments contain only a few memory accesses and thus the timeout affects the detection delay. For larger queue sizes enough instructions fit in a single segment that the log is usually filled before the timeout is reached. The exception to this is when programs feature large runs of instructions with very few loads and stores, for example bitcount. Without the timeout, very large segments of code appear, causing the maximum detection delay to increase significantly. However, a 50,000 instruction timeout is enough in this case to reduce maximum delay by 250× with no performance impact.

Number of Cores Figure 13 shows how performance scales across different numbers of cores devoted to error checking for the benchmarks. We see that N cores at a frequency of M (in MHz) is comparable in performance to $2N$ cores at a frequency of $\frac{M}{2}$. For example, 6 checker cores at 1GHz is comparable to 12 cores at 500MHz. This is expected given the identified parallelism in error detection.

In fact, a large number of cores at a low clock frequency outperform fewer at higher frequencies. This is because of the load-store log structure where only $n-1$ checker cores are in use at any given time, since at least one is always waiting for its segment to be filled unless the main core is stalled. This means that better utilisation, as a percentage of the total compute power of the cores, is achievable when more cores, and thus more load store log segments, are available.

B. Area Overhead

Publicly available data places the RISC-V Rocket, the closest available core in terms of size to the ones we propose for our checker units, at 0.14mm^2 area per core on a 40nm process [45]. In comparison, at 20nm, the Cortex A57 is

2.05mm^2 per core [46] excluding shared caches. Twelve E51-sized cores would therefore fit in approximately 0.42mm^2 combined at the same technology node.

The SRAM added for instruction caches, register checkpoints, load forwarding unit and the load-store log is 80KiB in total, which is approximately 0.08mm^2 area overhead [47]. Combined, this places the error detection hardware at approximately 24% area overhead compared to the original core without shared caches. When a 1MiB single-ported L2 cache at approximately 1mm^2 [47] is also included, the area overhead is approximately 16% of the original core.

This is a very approximate estimate: Rocket has a different ISA from the A57, and the out-of-order core we model is faster than an A57, increasing the number of checker cores required in our experiments (a real implementation would need fewer).

Still, it is clear that the overhead is massively reduced compared to dual-core lockstep, the current state-of-the-art, which doubles silicon area.

C. Power Overhead

Power information is more challenging to estimate. Models such as McPAT [48] are unable to account for the low power consumption of small cores of approximately 34μW per MHz [45] at 40nm, compared with 800μW per MHz for a Cortex A57 [46] at 20nm. Using twelve small cores and without scaling for feature size, we obtain a power overhead of approximately 16% for our system, showing that the impact is minimal. Since at 20nm the power consumption for the Rocket core will be less, this represents an upper bound, and we should expect the true value to be significantly lower still.

D. Bigger Cores

The out-of-order cores we simulate are relatively small, designed to mimic the behavior of conventional Arm systems. However, our technique extends favorably to larger main cores that are more aggressive, since these realise only a sublinear increase in single-threaded performance. Although we would need more checker cores, performance scales linearly with the power and area budget devoted to them due to the exploitation of thread-level parallelism. This means that relative overheads diminish significantly. Such cores may also feature simultaneous multithreading and for these each concurrent thread would be separated to a different checker core, but otherwise the scheme works similarly.

E. Summary

We have shown that twelve checker cores running at 1GHz are enough to have a performance impact of 3.4% maximum across a wide set of benchmarks, with mean error-detection times of 770ns. We have estimated the area and power overheads of the technique compared to an unchecked core at around 24% and 16% respectively, which are both significantly lower than existing techniques [3], [10], [12], [13], [49].

VII. Related Work

A. Lock-Stepping

There are many examples of hardware duplication for error detection, where copies of a program are run through identical logic and the results compared. This is currently used in the ARM Cortex R series of processors [9], as they are intended for high error and high reliability environments, such as cars and space. More recently, triple-lockstep designs, which perform majority voting to correct errors, have been developed [3]. Historically, similar techniques were used by the IBM G5 [10] and the Compaq Himalaya [49].

Mukherjee et al. [12] present a chip-level redundantly multithreaded scheme, where the second core trails the first in order to reduce cache misses. Gupta et al. [50] instead argue for duplication at a finer granularity, through multiple copies of individual pipeline stages in a fabric, rather than the more coarse-grained core duplication of industry schemes. This allows better tolerance of hard faults when errors are common. Hernandez and Abella [2] give a scheme to improve the detection delay for light-lockstep systems, where only some applications need error detection, and thus hardware can be repurposed if the second core is needed for detection.

B. Redundant Multi-Threading Hardware

Rather than a static duplication of hardware, many schemes have suggested using dynamic scheduling on processors featuring simultaneous multi-threading. AR-SMT [11] presents a redundant multi-threading scheme for fault detection, where two threads are run on the same processor. However, this does not cover hard errors, because the same hardware is used for both computation. In addition, it uses up a processor context that could be used for more computation, and comes at a significant performance overhead. Indeed, Mukherjee et al. [12] suggest that redundant multi-threading techniques come at a performance overhead of 32%. Schuchman and Vijaykumar [13] improve the ability of redundant multi-threading to detect hard faults by rearranging instructions within the trailing thread, altering the hardware resources used, at the expense of a further 15% performance degradation.

Reinhardt and Mukherjee [1] present the concept of a sphere of replication as it applies to redundant multi-threading: the parts of the system which are replicated. They further present the use of a load-value queue to forward results from the computation thread to the replication thread, instead of duplicating the page file as in AR-SMT [11]. This is similar to that used in our scheme. Smolens et al. [51] suggest the removal of this queue by noting that, in the common case, two threads will observe the same values from cache loads without explicit duplication, and instead use detection and recovery to correct any differences by treating them as errors, at the expense of performance.

Rashid et al. [14] utilise a similar form of parallelism to that which we exploit, to run error detection on a homogeneous multicore. The scheme pays a large area cost, but reduces energy usage by dynamic frequency-voltage scaling. We build on their insights by using a heterogeneous system to reduce area and energy further, and design an alternative forwarding system to increase parallelism and negate the need for a large L1 cache per core.

C. Software Schemes

It is also possible to provide error detection entirely in software, without hardware additions. Khudia and Mahlke [52] detect errors in software for soft applications, where only parts of the application are error-intolerant, such as video decoding. The significant overheads involved are reduced by only repeating computation for error-intolerant portions. Thomas and Pattabiraman [53] identify heuristics to select which parts of applications to check for high error coverage. Wang and Patel [54] provide a scheme for partial fault detection, by only responding to errors which trigger exceptions when they are not caught. Reis et al. [25] present SWIFT, a solution which duplicates instructions in the same thread to provide limited coverage of soft faults. Jeffery and Figueiredo [55] give a virtual lockstepping scheme, where a hypervisor is used to duplicate inputs and perform comparisons of multiple virtualised copies of an operating system. Veeraraghavan et al. [56] utilise a form of program slicing, as we use in our system, to solve a different but related problem: deterministic recording of execution for multicore workloads in software.

D. Hybrid Schemes

Hardware schemes suffer from a large cost in terms of silicon area, and software schemes suffer from a lack of coverage for hard errors, and high performance costs. To mitigate these, hybrid schemes have been proposed. For example, Reis et al. [24] present CRAFT, a combination of SWIFT [25], a software only scheme, and redundant multi-threading [1], [11], [12]. This uses compiler assistance to duplicate instructions, changing redundant stores to perform checks using a special hardware detection structure.

E. Heterogeneity

The use of heterogeneous cores for error resilience already has precedence. Ansari et al. [57] couple a lightweight core with a newer fast core. When the fast core begins to fail, it is used to provide hints, such as branches and loads, to the slower, functionally correct core, to reduce the performance gap. LaFrieda et al. [58] dynamically couple cores that can differ due to manufacturing defects, so that those which are faster are matched together to provide error detection, as are those which are slower or broken. DIVA [21], [22] adds in-order execution units towards the end of an unverified out-of-order pipeline to repeat computation and check data forwarding. These units run at the same clock speed as the rest of the core, and achieve parallelism by checking each instruction individually. This means ECC is required on all architectural state within the original processor to avoid communication errors, which is impractical in a high performance design.

F. Other Hardware Schemes

Other hardware fault tolerance schemes have been proposed, for example Clover [59], which uses hardware wave detection to detect cosmic rays hitting a system. Many schemes have been proposed to deal with retiring components efficiently once hard errors have been detected. Aggarwal et al. [60] partition multicore hardware into fault zones once errors have been detected, redistributing power dynamically based on how much of the core is still alive. Romanescu and Sorin [61] allow a fraction of the cores in a system to be used for spare parts at the pipeline granularity, to fix hard faults in a system. Gupta et al. [62] use a tiled web architecture which allows slow or broken pipeline stages to be weaved out. Powell et al. [63] allow the use of partially broken hardware by detection and migration of just the operations known to be faulty.

VIII. Conclusion

Current fault detection techniques are limited by high overheads, in terms of energy, silicon area, and performance. We have developed a technique to perform error detection for high-performance, out-of-order processors at low area, performance and energy cost by exploiting new parallelism in the redundant repetition of the program. Our scheme checks multiple parts of the execution simultaneously on a set of small cores embedded beside the main out-of-order CPU.

Evaluating over a wide variety of benchmarks, twelve small checker cores running at 1GHz give enough performance to limit average slowdown to 1.75% (maximum 3.4%). The mean error detection delay for each evaluated benchmark averages at 770ns, with 99.9% of all loads and stores checked within 5000ns, and all checked within 45μs: this is larger than with a lock-step system, but is more than offset by the reduction in chip area and power usage attainable, and is justifiable in the relevant domain spaces.

Future work will look at extending the scheme to perform correction of errors within a microprocessor, rather than just detection, to enable low-overhead complete fault tolerance.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/M506485/1, and Arm Ltd. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.21857.

References

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

[2] C. Hernandez and J. Abella, "Timely error detection for effective recovery in light-lockstep automotive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, 2015.

[3] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A triple core lock-step (TCLS) ARM®Cortex®-R5 processor for safety-critical and ultra-reliable applications," in *DSN-W*, 2016.

[4] M. Rausand, *Reliability of Safety-Critical Systems: Theory and Applications*. Wiley, 2014.

[5] M. Snir, R. W. Wisniewski, J. A. Abraham *et al.*, “Addressing failures in exascale computing,” *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, May 2014.

[6] A. Geist and S. Dosanjh, “IESP exascale challenge: Co-design of architectures and algorithms,” *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, Nov. 2009.

[7] D. Zhao, D. Zhang, K. Wang, and I. Raicu, “Exploring reliability of exascale systems through simulations,” in *HPC*, 2013.

[8] C. Turner, “Safety and security for automotive SoC design - arm,” http://www.arm.com/files/pdf/20160628_B02_ATF_Korea_Chris_Turner.pdf, 2016.

[9] N. Werdmuller, “Addressing functional safety applications with Arm Cortex-R5,” http://community.arm.com/groups/embedded/blog/2015/01/22/addressing-functional-safety-applications-with-arm-cortex-r5, 2015.

[10] T. J. Slegel, R. M. Averill III, M. A. Check *et al.*, “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, vol. 19, no. 2, 1999.

[11] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors,” in *FTCS*, 1999.

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

[13] E. Schuchman and T. N. Vijaykumar, “Blackjack: Hard error detection with redundant threads on smt,” in *DSN*, 2007.

[14] M. W. Rashid, E. J. Tan, M. C. Huang, and D. H. Albonesi, “Exploiting coarse-grain verification parallelism for power-efficient fault tolerance,” in *PACT*, 2005.

[15] A. R. Pargeter, “An example of strong induction,” *The Mathematical Gazette*, vol. 80, no. 488, 1996.

[16] https://www.sifive.com/products/coreplex-risc-v-ip/e51/.

[17] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “The impact of technology scaling on lifetime reliability,” in *DSN*, 2004.

[18] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, “Predicting the number of fatal soft errors in Los Alamos national laboratory’s ASC Q supercomputer,” *IEEE Transactions on Device and Materials Reliability*, 2005.

[19] S. Borkar and A. A. Chien, “The future of microprocessors,” *Communications of the ACM*, vol. 54, no. 5, 2011.

[20] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM errors in the wild: A large-scale field study,” in *SIGMETRICS*, 2009.

[21] T. M. Austin, “Diva: A reliable substrate for deep submicron microarchitecture design,” in *MICRO*, 1999.

[22] C. Weaver and T. M. Austin, “A fault tolerant approach to microprocessor design,” in *DSN*, 2001.

[23] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr, “Design and implementation of the POWER6 microprocessor,” *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, 2008.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, “Design and evaluation of hybrid fault-detection systems,” in *ISCA*, 2005.

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

[26] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *ISCA*, 2010.

[27] M. Shafique and J. Henkel, “Agent-based distributed power management for kilo-core processors,” in *ICCAD*, 2013.

[28] “Green 500,” http://www.top500.org/green500/lists/2017/11/, Nov. 2017.

[29] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, “Test-driving intel xeon phi,” in *ICPE*, 2014.

[30] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, “Evolution of thread-level parallelism in desktop applications,” in *ISCA*, 2010.

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

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

[33] International Organization for Standardization, “ISO 26262: Road vehicles – functional safety,” 2011.

[34] N. Werdmuller, “Addressing functional safety applications with ARM Cortex-R5,” https://community.arm.com/iot/embedded/b/embedded-blog/posts/addressing-functional-safety-applications-with-arm-cortex-r5, Jan. 2015.

[35] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, “Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery,” in *ISCA*, 2002.

[36] A. Jhingran and P. Khedkar, “Analysis of recovery in a database system using a write-ahead log protocol,” in *SIGMOD*, 1992.

[37] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *ISCA*, 1993.

[38] https://community.arm.com/processors/f/discussions/4503/lock-step-mode-execution-on-cortex-r5/11365#11365.

[39] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, “The hpc challenge (hpcc) benchmark suite,” in *SC*, 2006.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in *WWC*, 2001.

[41] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.

[42] N. Binkert, B. Beckmann, G. Black *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[43] A. Gutierrez, J. Pusdesris, R. G. Dreslinski *et al.*, “Sources of error in full-system simulation,” in *ISPASS*, 2014.

[44] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, “FTI: High performance fault tolerance interface for hybrid systems,” in *SC*, 2011.

[45] https://riscv.org/wp-content/uploads/2015/02/riscv-rocket-chip-generator-tutorial-hpca2015.pdf.

[46] http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6.

[47] M. Yabuuchi, Y. Tsukamoto, M. Morimoto, M. Tanaka, and K. Nii, “20nm high-density single-port and dual-port srams with wordline-voltage-adjustment system for read/write assists,” in *ISSCC*, 2014.

[48] 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.

[49] A. Wood, “Data integrity concepts, features, and technology,” Tandem Division, Compaq Computer Corporation, White Paper, 1999.

[50] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, “The stagenet fabric for constructing resilient multicore systems,” in *MICRO*, 2008.

[51] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, “Reunion: Complexity-effective multicore redundancy,” in *MICRO*, 2006.

[52] D. S. Khudia and S. Mahlke, “Harnessing soft computations for low-budget fault tolerance,” in *MICRO*, 2014.

[53] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *DSN*, June 2013.

[54] N. J. Wang and S. J. Patel, “Restore: Symptom based soft error detection in microprocessors,” in *DSN*, 2005.

[55] C. M. Jeffery and R. J. O. Figueiredo, “A flexible approach to improving system reliability with virtual lockstep,” *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, 2012.

[56] K. Veeraraghavan, D. Lee, B. Wester *et al.*, “Doubleplay: Parallelizing sequential logging and replay,” in *ASPLOS*, 2011.

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

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

[59] Q. Liu, C. Jung, D. Lee, and D. Tiwari, “Clover: Compiler directed lightweight soft error resilience,” in *LCTES*, 2015.

[60] 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.

[61] 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.

[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] 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.