ParaMedic: Heterogeneous Parallel Error Correction

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

Abstract—**Processor error detection can be reduced in cost significantly by exploiting the parallelism that exists in a repeated copy of an execution, which may not exist in the original code, to split up the redundant work on a large number of small, highly efficient cores. However, such schemes don't provide a method for automatic error recovery.**

We develop ParaMedic, an architecture to allow efficient automatic correction of errors detected in a system by using parallel heterogeneous cores, to provide a full fail-safe system that does not propagate errors to other systems, and can recover without manual intervention. This uses logging to roll back any computation that occurred after a detected error, along with a set of techniques to provide error-checking parallelism while still preventing the escape of incorrect processor values in multicore environments, where ordering of individual processors' logs is not enough to be able to roll back execution. Across a set of single and multi-threaded benchmarks, we achieve 3.1% and 1.5% overhead respectively, compared with 1.9% and 1% for error detection alone.

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

I. Introduction

Fault tolerance is an increasingly important property for computer processors. As transistors shrink, both hard (permanent) and soft (transient) faults become more common in silicon chips [1], [2], [3], as a result of increased variability, lowered energy required to cause a transistor to flip bits from, for example, cosmic rays, and increased points of failure [4]. In addition, fault-intolerant computation is becoming increasingly important. Automotive applications require strict safety standards to be met, and thus typically require fault detection and correction for all safety-critical components [5], [6], [7]. With the advent of self-driving cars, the performance requirements make traditional fault-tolerance schemes, such as lock-stepping [6], impractical, as they double silicon area and CPU power consumption. Similarly, many scientific and large-scale applications are increasingly error intolerant due to the number of potential failures in such systems.

A recent innovation in this area is heterogeneous parallel error detection [8]. This allows error detection to be achieved at a significantly reduced power-performance-area (PPA) overhead compared with previous schemes, by exploiting new parallelism that exists in code that has already been executed once, to allow the checking to be done on parallel checker cores, each of which is orders of magnitude smaller and lower power [9], [10] than a traditional out-of-order superscalar [11]. However, previous work does not extend this to cover correction of faults in a system, and so errors are allowed to propagate outside the sphere of replication, and returning to a safe state must be achieved by manual intervention. Indeed, there are a variety of factors that make this challenging. The increased latency of error detection for such schemes when compared with lock-stepping and redundant multi-threading, necessary to exploit parallelism, makes rolling back incorrect writes in the correct order a challenge. Keeping track of unchecked state in an efficient manner is also difficult: handling this in software is too expensive, as anything that has been written potentially may need to be rolled back. Hard errors are challenging to correct, as the hardware running the application will repeatedly exhibit the same error, and the requirement of checking for errors before the result is propagated out of the system, necessary for correct behaviour outside of a sphere of replication [4], means high checking latency may be impractical for performance.

We develop ParaMedic, a parallel error-correction technique to solve these issues for both single and multi-core processors, and make hard- and soft-error correction in hardware both practical and highly efficient. To extend heterogeneous parallel error detection [8] to full correction, we use solutions inspired by transactional memory [12]: fine-grained eager versioning to support efficient rollback from many different concurrent checkpoints, combined with coarser-grained lazy versioning to create a total order on rollback between multiple parallel cores. To maintain correct behaviour even in the presence of hard-errors, we develop a novel hard-fault log to guarantee forward progress. We can provide error-correcting codes for correct rollback by reusing resources from error detection [8] and the existing cache infrastructure. And we can dynamically extract the maximal amount of error-correction latency and parallelism from an application while still allowing correct executions to propagate out of their fault domain quickly, by dynamically adjusting checkpoint frequency using an additive-increase multiplicative-decrease technique.

Across a set of single and multi-threaded benchmarks [13], we achieve 3.1% and 1.5% overhead respectively, compared with 1.9% and 1% for error detection alone.

II. Background

A. Dual-Core Lockstep

Current techniques for reliability in commercial systems tend to involve dual-core lockstep [6], [14]. The same code is run on two identical copies of a processor, usually with some delay to avoid correlated errors, and the results compared. This approximately doubles core area and power consumption, as everything has to be run twice in the same way.

A dual-core lockstep detection system can extend to correction by adding a further processor [6], where a majority vote is taken on instruction commit. However, overheads become significantly worse, now requiring three times the core area and power of an unprotected system. For this reason, academics have considered schemes based on additional redundant threads instead.

B. Redundant Multi-Threading

In redundant multi-threading [4], [15], [16], [17] the same code is run twice within two different threads on the same core, typically with hardware forwarding of load and store values, to check correctness. This suffers from a significant reduction in performance compared to no checking, greatly increasing energy consumption, and is unable to detect hard faults without introducing spatial diversity [18], as the same hardware is used for both the check and the original execution. Since threads are decoupled from each other, errors cannot be caught before instructions retire, and so can propagate into main memory [16]. This means that typically software or hardware checkpoints are used to revert to a correct state. However, it is possible to couple the threads more tightly [19] to ensure the checker thread executes before the main thread commits, though this heavy restriction in scheduling decreases performance even further.

C. Heterogeneous Parallel Cores

A recent alternative architecture for error detection appends a set of small, power-efficient cores to a main high-performance core to perform the redundant computation in parallel [8]. The key insight is that running code a second time to check its correctness is more parallel than the original execution. It is possible to split up the application by taking periodic register checkpoints, then overlap the checking of the code between multiple checkpoints. Due to the parallelism available in the second run, a set of simple cores (the checker cores) can together provide enough computational power to keep up with a high-performance core.

To allow the checker cores to replay load values, and check store values and load and store addresses, all loads and stores are extracted in-order from the program stream at commit time within the large out-of-order processor used as a main core. These are placed into a load-store log, which is partitioned such that it is divided equally between the checker cores. When a segment is filled, or an instruction timeout is reached, a new checkpoint is triggered, and a check between the previous checkpoint and the new one is started on the corresponding checker core. Stores are allowed to propagate to main memory before they are checked, to avoid impacting performance significantly.

This approach to error detection is highly advantageous, with performance, area and power overheads of 1.8%, 24% and 16%, respectively compared to a processor without checking [8]. When compared with more conventional lock-stepping schemes [6], the area and power requirements are reduced significantly from the 100% cost in both dimensions caused by doubling the core. This makes heterogeneous parallel error detection the most suitable starting point for an error-correction scheme.

D. Challenges for Error Correction

Although at first glance it seems straightforward to augment a parallel error-detection scheme with circuitry for correcting errors, there are a number of challenges that present themselves. Tight coupling of parallel error-detection circuitry with the main core is infeasible because a large number of instructions need to be executed, without being checked, to achieve parallelism in the detection. We must therefore design a system that can tolerate much more latency between error initiation and detection, without sacrificing correction ability. This means we must be able to log a large number of potentially incorrect stores to be able to roll them back. It also means that we need to be able to deal with the complexities of multicore shared memory, where errors that remain uncaught for long periods of time may propagate around the system. Still, there is also an opportunity because, unlike in dual-core lockstep or redundant multithreading, many copies of hardware capable of executing code exist, as detection is achieved via multiple checker cores that have the same capabilities as the main core. This means that a majority vote can be achieved without tripling or even doubling the hardware overheads, and so even hard errors can be corrected efficiently. We take these challenges forwards in this paper to design ParaMedic, a parallel error-correction architecture, considering the requirements for a single core setup (section III) before extending to multicore (section IV).

III. Single-Core Correction

Figure 1 shows an overview of ParaMedic, our system for heterogeneous error correction. The hardware we add for single-core correction is coloured green; structures for multicore correction (described in section IV) are orange with hatching.

Execution proceeds in the same manner as with heterogeneous error detection, with the following changes. Addresses and data for loads and stores are placed into the load-store log in program order, along with ECC to protect them. Each segment of the load-store log obtains a timestamp each time it starts being filled, so as to create an ordering between segments. Once full, the associated checker core starts validating the segment's contents by re-executing all instructions. In the common, error-free case, the segment can be reused once all segments with earlier timestamps have been successfully validated. However, on detection of an error, execution is stopped and state reverted to the register checkpoint at the start of the erroneous log segment by rolling back each load-store log segment in reverse sequential order, so as to "undo" each of the stores that has taken place up to that point. The main core then starts running the application again, starting from this checkpoint.

The following sections describe these operations and the extensions we require to achieve this form of execution.

A. Partitioned Load-Store Undo Log

As in the scheme for error detection only [8], a hardware SRAM log records past loads and stores, so that the smaller checker cores can replicate load data, and check store data and load and store addresses. This is partitioned, with separate

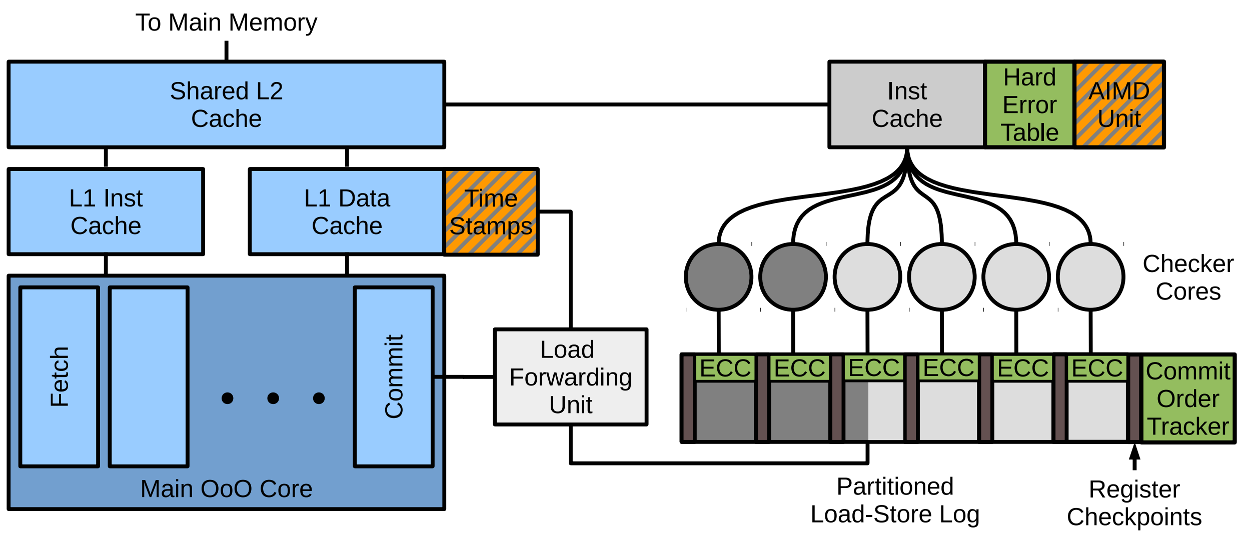


Fig. 1: ParaMedic: a heterogeneous parallel error-correction system. Units required for single-core error correction are presented in green. Units for multi-core correction are presented in orange with hatching. Structures for necessary for detection of errors are in grey, and the main out-of-order core in blue.

segments of the log for each checker core. We store virtual addresses stored in the load-store log, as this removes the need for translation by the checker cores. Writes are sent to the load-store log after having ECC for the cache calculated on them, so that any errors before this will propagate into the log and be caught, and any on the separate cache path will be corrected by ECC bits. Load data is duplicated by the load forwarding unit before being forwarded, and errors within log forwarding cause errors to be detected on future checks.

To enable correction of errors in the system, we must be able to revert writes that may be incorrect, since they occur after a check fails. Unchecked data that is potentially incorrect propagates into the cache system, to allow efficient forwarding of writes to new computation. We therefore take a copy of the old value of each word written to the L1 cache and record it in the load-store log, so as to provide the ability to undo stores. We do this for every write, extending the amount of data stored per write compared to error detection alone. The data fields recorded for each load and store are shown in figure 2. We also add a dedicated load/store bit, for the unroller to determine which log segments are loads or stores, which the detection mechanism infers from the instruction stream. On detection of an error, these writes are then rolled back by walking the log in reverse order, and writing the old values back to the cache. The virtual addresses in the load-store log are retranslated by the TLB upon a rollback, moving the translation to the uncommon, rather than the common case, as translation does not need to be performed for correct segments.

B. ECC

Error detection by itself does not require ECC in the load-store log. Load data forwarded to the load-store log from the cache takes a separate path from the connection to the main core's load-store buffer, so there is never a point where there is a single copy of unprotected data. Errors in addresses or data held in the log will be detected, because the error occurs after the original execution and the two will diverge. This also holds true for error correction in the case of load and store data. However, if an error were to occur in the store address or the old data values (those overwritten by the store), we would either write to an incorrect address or write incorrect data in the event of a rollback. For correction, therefore, we must protect store addresses and old data values with ECC.

To avoid recomputation of ECC bits [20] for the data values, it is useful to recover as much of this information as possible from the cache. We assume that ECC bits are stored per word in the cache [21], [22], and copy those directly. While the cache will have ECC bits for the address tag in the cache, these are likely to represent the physical address, and cover only line rather than byte granularity, and so we calculate a new ECC for the virtual address on every store. This ECC data is then stored in the load-store log (see figure 2).

C. Commit Ordering

Once a checker core has finished execution of its load-store log segment, it validates the register checkpoint at the end of the segment (and the beginning of the next). Depending on how many instructions are in each segment, and the types of instruction, these checks can complete out-of-order. If we are only detecting errors, a checker core's log segment can be reallocated immediately after a check is complete, because if the check is successful, the data is no longer needed. However, when correcting errors, we can only allow this to occur once we are certain that there are no errors in earlier segments (i.e., those containing older program instructions), which may take more time to check if they contain more or longer-latency instructions, for example. If earlier log segments do contain errors then we need to use the current segment to roll back

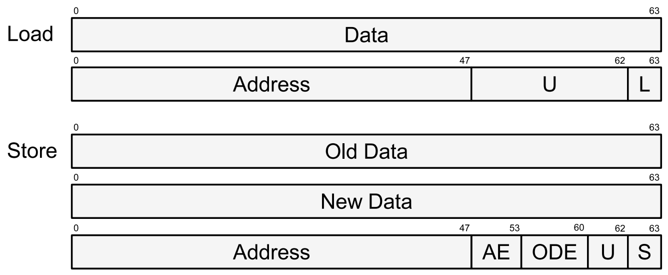


Fig. 2: Structure of entries in the load-store log. Loads use two 64-bit entries; stores use three. We assume that ECC bits give single error detect, single error correct (SEDSEC) capability, using Hamming codes [20]. Abbreviations are Address ECC (AE), Old Data ECC (ODE), Unused (U), Load (L) and Store (S). Load and store bits are placed such that they are the first bit seen in a packet when read in reverse on an unroll.

stores, so as to ensure that memory values are correct for the register checkpoint we are reverting to. We can therefore only reallocate a load-store log segment once all prior segments have been successfully validated.

The cases in which this new constraint can affect performance are those where utilisation of the checker cores is reduced, and thus the main core has to be stalled for longer, waiting for load-store log space. This is only significant when the runtime length of each check is highly variable. By keeping them at a similar instruction length in any given phase, the additional slowdown can be minimised.

D. Register Checkpoints

For detection alone, for each segment of the load-store log, we must store a checkpoint of the register file as the main core saw it at the end of that particular sequence of instructions. If this matches what the checker core produces, then there is no error. However, to perform error correction, we must store and be able to roll back to the checkpoint at the start of a segment (i.e., the checkpoint at the end of the previous segment). This increases the length of time we must keep the starting register checkpoint before overwriting it—essentially the previous segment's register checkpoint cannot be overwritten until the following segment commits. However, the previous segment's undo log can still be filled while the current segment is being checked, and so no slowdown is caused by this: if the main core must stall waiting to be able to overwrite a register checkpoint, it would also stall waiting for the subsequent load-store log segment to become free.

ECC is unnecessary on register checkpoints while the associated partition is being checked. This is because information redundancy is achieved by the two execution copies, one from the main core and one from the checker core. Still, since we may need to roll back to a register checkpoint after it has been checked, due to a later error, a window of vulnerability is introduced. We can mitigate this with ECC on the register checkpoints but, crucially, this is only necessary once the register checkpoint has been validated. This means the ECC can be calculated in parallel with the computation check, does not need to be calculated when the checkpoint is taken, and does not sit on any performance-critical path.

E. Error Recovery

On the detection of an error, ParaMedic must roll the system back to a consistent state. We do this by stopping the main core, stopping all checker cores with timestamps that are later than the one exhibiting the error, then rolling back all filled and partially filled load-store log segments, starting with the youngest (i.e., most recent instructions), in reverse order, until the segment with the error has been rolled back. The register file of the main core is then restored to that at the start of the segment with the error. By storing a read/write bit in the log for each entry (figure 2), which is unneeded for the initial check as it can be determined from the instruction stream, we can quickly identify old write data.

As there may be multiple writes to the same location, we must perform this in sequential reverse order to regain a consistent state. While this is a slow operation, it occurs infrequently, as hard and soft errors are relatively uncommon. Further, we can place bounds on it by altering the number of instructions per log segment accordingly, at the expense of greater checkpoint overhead for smaller log segments. Since the load store log stores virtual addresses, to avoid retranslation during checking, we assume the TLB and page table walkers are protected using their own redundancy mechanisms such as ECC, to prevent the need for retranslation in the common case and reduce the amount of translation logic required. This means that, for a rollback, translation must be repeated to re-retrieve the physical address. Any changes in translation map state between execution and recovery will naturally be rolled back by reversing all subsequent writes.

Events such as exceptions are speculatively handled by the main core, as with all other computation, before checks are completed. If this exception later turns out to be incorrect (for example, a soft error changes a value which causes an out-of-bounds memory access), the exception handling is naturally rolled back by the undoing of all loads and stores.

An earlier check that hasn't yet completed may fail after a later one in program order, particularly when we have hard faults in a system. In this case, we must wait for the future rollback to complete before proceeding to roll back further. On a failure, we increment the current commit timestamp, as opposed to returning to the timestamp before the error occurred. This, as we shall see later (sections III-F and IV-A), reduces the amount of book-keeping in the cache without affecting correctness, and enables hard-fault tracking. We mark all checkpoints after the error as "rolled back", equivalent to "committing", save for the fact that no new roll back is necessary on new errors, but do not commit them until all previous checks have completed. This ensures correct behaviour even if an earlier error is later detected. An example of this behaviour is shown in figure 3.

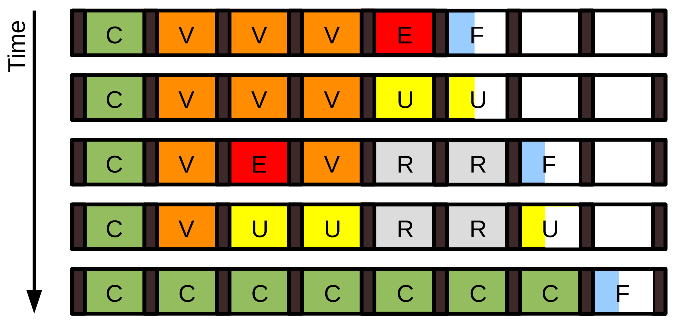


Fig. 3: *Example of an error being rolled back, followed by discovery of an earlier error. Though it would not harm correctness to repeat roll back of entries marked as "R", treating them as committed prevents redundant work. Abbreviations are Committed (C), Validating (V), Error (E), Undoing (U), Rolled back (R), Filling (F).*

		Committed Error		Speculative Error		
Time	Committed Time	Data	Addr	Time	Data	Addr
0	0	0	0	0	0	0
7	1	0	0	5	0x43	0xA4
8	1	0	0	3	0x84	0xBE
8	7	0x84	0xBE	3	0x84	0xBE
11	7	0x84	0xBE	9	0x84	0xBE

Fig. 4: *An example of hard error tracking, using the execution of figure 3 as a starting point. A speculative error occurs at time 5, and is checked against the current committed error, but does not match. When the earlier time 3 error is checked, this replaces the value in the speculative-error register. Finally, when time 3 is committed, this data goes into the committed-error register. The program is re-executed, and the same error is detected, triggering hard error recovery and maintaining forward progress.*

F. *Hard Faults*

Transient errors are unlikely to occur twice in the same place, so we can successfully recover from them by returning the system to a consistent state and beginning execution again with associated checking, which should succeed the second time. However, this is not the case with hard faults, either in the core itself or within the checker hardware of a core, where the same error will be detected twice. Further complicating this is that the same hard fault may manifest in different locations in a checker log even if we assume the log is in program order, due to non-deterministic scheduling. This makes a hard fault difficult to distinguish from a soft fault.

Our solution aims to maintain forward progress in the presence of hard faults by keeping track of the address/data pair of the most recent error detected in a previous-error log. The address is either a memory location or an architectural register, depending on whether the fault was detected in the register checkpoint or within a memory instruction. On detection of an error we check this pair and if there is a match then we move to a different checker core for the subsequent re-execution after rollback. If that doesn't succeed in correcting the issue, we can then move to different main core and checker unit for the next attempt.

To succeed in detecting hard faults, this scheme ensures the error written back to the tracking hardware is the first error in a sequence. Before we update the previous-error log, all prior checks must have been completed. We do this by keeping both a speculative-error register and a committed-error register. The former is updated whenever an error occurs that has an earlier timestamp than the existing error. The latter is updated when the committed timestamp goes past the timestamp of the speculative-error register. All new errors are compared with the committed-error register. This won't necessarily identify all hard errors as being permanent, but is sufficient to maintain forward progress. As the timestamp is monotonically increasing, the committed-error register is updated correctly even after rollbacks. An example of this is shown in figure 4.

If no other main cores are available we can still recover from hard faults by effectively checking the code on two different checker cores. We store the data and address of the committed error, using the value from the first checker core. We then restart execution, ensuring the new start point is scheduled for checking on a different core. If the next error matches, it is assumed that the main core is incorrect, and the checker cores correct. The program is rolled back to immediately before the error occurred, the correct operation proceeds, and execution is continued on the main core, with the register file copied from a checker core immediately after the operation.

G. *Context Switches*

As with all interrupts, context switches trigger new checkpoints, to avoid replaying them in the middle of a check. On a rollback, this behaves correctly: the stores are rolled back for the process that has been switched to, followed by the stores for the earlier process, as a result of the timestamps for each core having a total ordering regardless of process.

If we have to roll back an incorrect computation, on the successive attempt at running the computation a context switch may be triggered at a different point. However, this should cause no issues, as the second execution is checked independently of the first failed attempt.

One complication with rolling back stores is that the load-store log structure tracks store addresses in the virtual address space, to avoid having to translate before checking. This means that to write back to the correct physical location, the rollback hardware must have access to the TLB, and also we must store a process ID per partitioned load-store log segment. As we switch entry on a context switch, we only have to store a single process ID for each segment.

H. *Kernel Code*

In addition to user-space code, ParaMedic must also check kernel-level code to maintain correct execution. This is also separated from user-space code by register checkpoints that are taken when any transfer to kernel mode occurs (e.g., system calls or hardware interrupts). Kernel-level registers must also be stored in the necessary checkpoints in this case. The error detection process works similarly, as all data the kernel-level code must read will also be forwarded in the load-store log. While checker cores themselves do not need access to kernel memory while performing this check, as loads and stores are forwarded, kernel-level code rollback does need to be able to write to kernel memory. As data may have propagated to the kernel code from unprivileged code and vice versa through, for example, system call arguments, we roll back both kinds of checkpoint in the detection of an error in either. In effect, we treat the two identically.

I. Writes Outside the System

To avoid having to report errors and corrections to other systems, a fault tolerant system should only allow correct results to escape from its sphere of replication. We therefore need to make sure that all data that leaves the system has been checked. Similarly, all unrepeatable reads and writes (for example, to some IO devices) must be non-speculative in terms of errors. To do this, we must stop the system, issue a check, and wait until the check has completed before issuing communications with other systems, or more general unrepeatable reads and writes. We consider all uncacheable data as coming from an unrepeatable operation and force checks to complete before the operation succeeds.

This could introduce a significant delay in the execution of such operations, when the number of instructions between checkpoints is large. However, if they are infrequent, this cannot cause a significant performance loss. If they are frequent, then each checkpoint will be small and the latency between each operation will thus be similarly small. Performance will in effect be limited by the checker cores, as the enforced checkpoints will cause every instruction between nonspeculative reads/writes to be in the same checkpoint. However, since such code is likely to be IO-bound, this is unlikely to reduce performance significantly, as the smaller cores will likely be able to keep up with larger ones. We have two cases: buffered IO only results in overhead at the single point of the DMA request, rather than on all IO operations, and unbuffered IO is not compute bound so can be adequately serviced despite lacking parallelism in ParaMedic in this scenario. Still, we come up with a scheme to ameliorate these issues, and others, using dynamic checkpoint scheduling, in section IV-D.

J. Summary

This section has dealt with the properties of ParaMedic necessary to develop a heterogeneous parallel error-correction scheme, under the assumption that a system has a single main core. The load-store log must be extended to include overwritten data, so we can roll back writes on detection of an error. These old values must be covered by ECC, to make sure rolled back values are correct, but most of this data can be recovered from the memory system. A stricter commit ordering needs to be placed on the checkpoints taken for each parallel check, to allow sequential rollback of data. We can correct hard faults by tracking the repetition of error observations, to guarantee forward progress, and to give the illusion of a fault-free system to the outside world, IO operations are delayed until all previous instructions are checked. The next section deals with error correction for multicore CPUs.

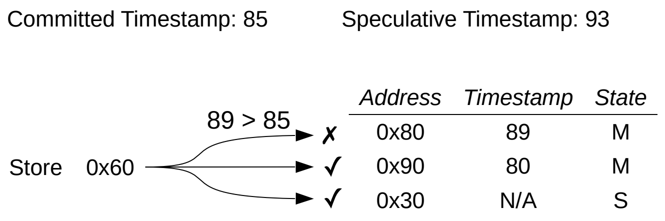


Fig. 5: An example store into the L1 cache. At the current timestamp, 93, a write miss occurs. We cannot evict the line at address 0x80 because its timestamp (89) is more recent than the committed timestamp (85), and so potentially-incorrect data could be written out to the L2 cache.

IV. Multicore correction

The solution presented in the previous section works for single core processors. However, when multiple main cores are included in a system the problem of correcting errors becomes more complicated because we have to know far data has propagated when committing and on an error.

As an example, suppose a check fails and ParaMedic rolls back to an earlier checkpoint. How do we know which other cores have seen data affected by the rollback, since they will also need to be rolled back to a correct state? Conversely, how do we know that all data used by a particular load-store log segment has been validated, so it is safe to commit the current entry? What ordering should we apply when two load-store log segments read data generated by the other? How do we know the order in which stores should be undone when we have multiple load-store logs being written to concurrently?

We solve all these issues by mandating that data must be checked before communication can occur. If a core cannot see uncommitted data from another core, errors are independent, and failed checks can be rolled back independently. However, we need to minimise the overheads of enforcing checks-before-communication. The following section discusses techniques used to achieve this, while still allowing high performance even for workloads with communication between cores.

A. Cacheline Timestamps

To prevent data propagation to other cores until it has been checked, we keep modified data within private caches until it has been committed. A timestamp, local to each core rather than to each process, is stored per cacheline in the L1 cache and data is only allowed to leave if its line hasn't been modified or if the line's timestamp is less than or equal to the most recently committed timestamp. This means that modified data cannot be evicted through the cache's replacement policy or coherence requests (either invalidating the line or changing to shared state). An example is given in figure 5. As uncommitted data cannot escape to shared caches, we do not need to extend this scheme further than the L1, reducing the storage overhead.

If a check fails, we roll back and undo stores by overwriting new cached data with its old value, leaving the timestamp unchanged (i.e., we do not undo the timestamp update that occurred with the store that is being reverted). As the failed checks are marked as ready to commit, but are not committed, this behaves correctly if an earlier check subsequently fails. When all previous checks have been completed, the timestamp will be below the last committed timestamp, and can thus be considered committed. This is a safe overapproximation: we may falsely mark data that was written before any uncommitted timestamps as uncommitted, however we still guarantee forward progress, as once all previous timestamps have committed, execution will be able to continue.

B. Data Eviction

To prevent unchecked data escaping the L1, we can only evict data once it has been committed. In cases where the data must be evicted, we pause the main core until the data has been checked. If the timestamp of the data is the same as that currently executing on the core, we immediately issue a checkpoint to start checking the data, to avoid deadlock.

As the core must stall at this point, we wish to do it as infrequently as possible. We thus bias against evicting uncommitted data by favouring eviction of other data within the cache set first. We stop the core if all cache lines within a set are uncommitted and modified, so one must be evicted.

A 36KiB load-store log can hold between 1,500 and 4,500 entries, approximately, assuming 64-bit words and depending on the ratio of loads to stores in the log. By comparison, a typical 32KiB data cache can store approximately 4,000 64-bit words, or 500 cache lines. However, multiple accesses to the same location take up multiple entries in the load-store log. Thus, provided there is some temporal or spatial locality, uncommitted data is unlikely to be evicted from the L1 as a result of a capacity miss. However, conflict misses in low-associativity caches are possible. These can be mitigated by increasing the associativity of the cache, or by using an eviction buffer for uncommitted writes, which is stored in order of timestamp.

C. Coherence Requests

In addition to eviction by other data and through invalidation, requests from the cache coherence protocol can also force data to be written to either the upper level caches, or the private caches of other cores. If a request is observed to a cache line that is currently uncommitted, the response must be delayed until the relevant check completes. This means that writes become visible in commit-order of checkpoints. If the write timestamp in the local cache is the same as the timestamp currently executing on the main core, again it is necessary to issue an early checkpoint. The coherence request further takes precedence over future writes to the cache-line and so a write-attempt by a core to a coherency-requested cache line must stall until the coherence request is satisfied. This is sufficient to avoid deadlock within the protocol: any cache lines that have been requested by another core will eventually be checked and released, as no cyclic dependencies can be constructed. If a check fails, and a cache line is rolled back, then the earlier data is available to the requesting core, once all checkpoints earlier than the failure are committed.

D. Variable-Length AIMD Checkpointing

Fundamentally, parallel error checking exploits increasing checking latency to extract parallelism in the redundant execution of code. However, in cases where we must stall a main core, or delay a coherence request, to issue early checks (e.g., unrepeatable reads and writes, cache evictions), we have overestimated the amount of latency the system can tolerate. One solution to this would be to reduce the number of instructions, loads, and stores allowed between each checkpoint. However, in cases where this behaviour does not occur, this would reduce performance, as taking register checkpoints is relatively expensive.

Our solution is to make the number of instructions allowed between each checkpoint (i.e., in the corresponding load-store log segment) dynamic. While we cannot do anything about overestimating the checking latency the first time this occurs, we can use this as a prediction that the same will happen in the short term future, and thus reduce the checkpoint length accordingly. To do this, while generating a stable estimate for checkpoint length, we use an additive increase multiplicative decrease (AIMD) scheme, as used in TCP [23]. Here, every event that triggers a pause in execution, or a delayed coherence response, halves the target number of instructions in a segment. In contrast, any checkpoints for which such events do not occur increase the target instruction limit by 5, up to a limit of 5,000, which is the maximum size of any load-store log segment. This avoids penalising the cases where such pauses do not occur, but is highly reactive when they do, thus rapidly decreasing the target length when necessary, and increasing the target slowly, to keep each checkpoint at a similar length in the short term.

E. SMT Extension

Our current discussion has assumed that caches are private, and only one thread can be executing on a core at a time. This does not hold true for cores that support simultaneous multi-threading [24], where instructions are committed from multiple threads at once. Here, we need a more sophisticated scheme that prevents loads and stores from different contexts from entering the same load-store log segment, stops unchecked writes from propagating to other threads, and defines an order in which writes should be reversed on an error.

ParaMedic separates committed loads and stores by thread ID, and issues separate checks to checker cores for each thread. Thread ID bits are concatenated to the timestamp: a check can commit when all previous timestamps with the same thread

Main Cores	
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 Branch Pred.	2048-Entry local, 8192-entry global, 2048-entry chooser, 2048-entry BTB, 16-entry RAS
Reg. Checkpoint	16 cycles latency
Memory	
L1 ICache	32KiB, 2-way, 1-cycle hit lat, 6 MSHRs
L1 DCache	32KiB, 2-way, 2-cycle hit lat, 6 MSHRs
L2 Cache	1MiB shared, 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, 5000 inst. max length
Cache	2KiB L0 ICache per core, 16KiB shared L1

TABLE I: Core and memory experimental setup.

Single Core Benchmarks	*Source*	*Input*
randacc	HPCC [25]	100000000
stream	HPCC [25]	
bitcount	MiBench [26]	75000
blackscholes	Parsec [13]	simsmall
fluidanimate	Parsec [13]	simsmall
swaptions	Parsec [13]	simsmall
freqmine	Parsec [13]	simsmall
bodytrack	Parsec [13]	simsmall
Multicore Benchmarks	*Source*	*Input*
blackscholes	Parsec [13]	simsmall
canneal	Parsec [13]	simsmall
fluidanimate	Parsec [13]	simsmall
swaptions	Parsec [13]	simsmall

TABLE II: Summary of the benchmarks evaluated.

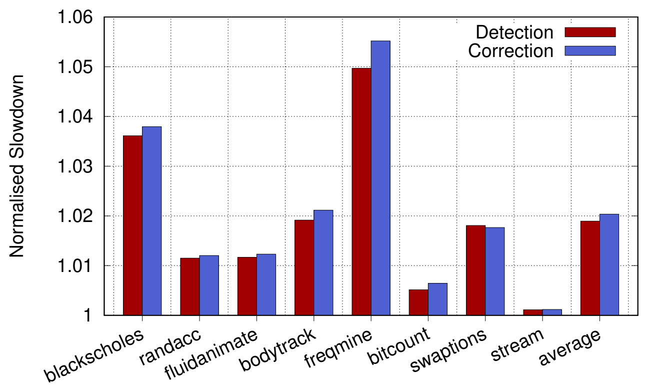


Fig. 6: Normalised slowdown for each single core benchmark, with both detection-only and correction schemes.

ID have committed. When a cache line is accessed that was modified by another thread at an uncommitted timestamp, we stall the reading thread, issuing an early checkpoint on the thread that wrote to the cache line if necessary, until the writing thread commits. This enforces a serial, independent ordering on writes between threads, so on an error we need only roll back the thread in which the error occurred.

F. Summary

This section has dealt with extending ParaMedic's error correction to multicore systems, while still allowing the undoing of stores after detecting an error. We avoid ordering issues by ensuring that any communication between cores only propagates correct data, by buffering writes in private caches based on their timestamps. If a coherence request forces communication early, the data response is delayed until after it has been checked, potentially triggering a new checkpoint. To prevent a loss of performance, the size of checkpoints is dynamically adjusted, based on the amount of communication and uncommitted data stored in the L1 cache, to adapt the coarseness of parallelism to suit the application.

V. Experimental Setup

To evaluate the performance impact and detection latency of ParaMedic, we modeled a high-performance system using the gem5 simulator [27] with the ARMv8 64-bit instruction set, and configuration given in table I. This is similar to systems validated in previous work [28] and close to that used in previous work on heterogeneous error detection [8].

The benchmarks we evaluate are given in table II. Where possible we choose benchmarks similar to those used in previous work [8]. However, as their scheme has no impact on the efficiency of multicore systems, unlike ours, due to the prevention of error propagation (section IV), we further add multi-threaded versions of Parsec [13] applications running on two threads. This required using gem5's threading module, m5threads, and thus ARMv7 instead of ARMv8. The subset of benchmarks chosen are those that run successfully using m5threads on ARMv7.

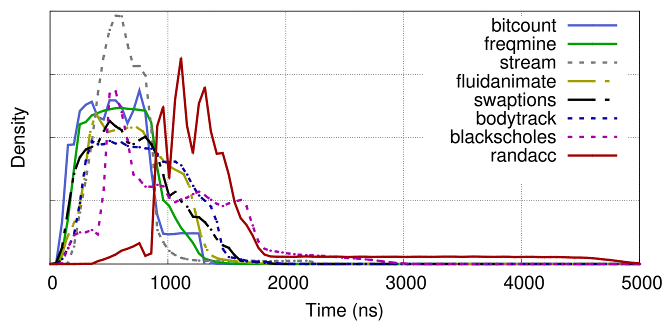
VI. Evaluation

Exploiting heterogeneous parallelism for multicore error correction incurs performance overheads of just 3.1%. We first look at the impact of the changes required for single core machines, as described in section III, showing minimal difference in performance compared to detection only. We then evaluate the techniques required for correctness on multicore machines.

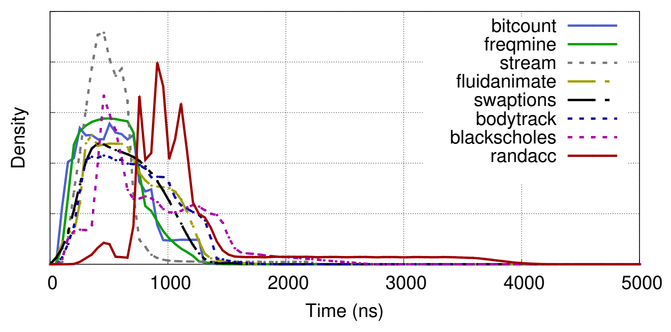
A. Single-Core Error Correction

Figure 6 shows normalised slowdown for a set of single core benchmarks, using both error detection [8] and the basic correction scheme suitable for single core systems from section III. In effect, the latter is error detection but with extra data for stores in the load store log, thus reduced maximum capacity, in-order commit of error-detection checkpoints, and blocking on IO requests to allow error checking to catch up.

We see that these cause very little extra slowdown compared with the baseline with no error detection. In many ways this is unsurprising. Increasing the amount of data in the load store log slightly increases the frequency of register checkpointing,



(a) Detection



(b) Correction

Fig. 7: Density plot to show delays between a store committing and being checked, with error detection and correction, respectively.

but not enough to have a large performance impact. Because checkpoints are typically similarly-sized, due to being either an equivalent number of loads and stores, or an equivalent number of total instructions if the limit is reached before a load-store log segment is filled, error-detection checkpoints usually commit in order, and so allowing out-of-order commit rarely improves resource availability. While the workloads we look at do feature IO operations, high-performance applications typically buffer these in main memory, and thus there are few operations where we have to stall and wait for detection to finish. Even if the latter were not the case, direct IO operations are sufficiently slow that each small checker core would be able to keep up with the main core, and thus the lack of exploitable parallelism is unlikely to affect performance.

Figure 7 shows the delay between an error occurring and it being found in both schemes. We see that the constraints introduced for correction actually reduce the delays observed slightly. This is typically because checkpoints are smaller, due to the larger amount of stored data, reducing the number of loads and stores in each checkpoint and causing each checkpoint to be issued and checked more quickly.

B. Multicore Error Correction

Next we evaluate the additional constraints introduced by section IV. The most salient of these is the prevention of data escaping from the L1 cache before it has been checked by a checker core. First, we evaluate the performance overhead for single-threaded benchmarks where we assume that data may be shared with other cores, along with the AIMD technique (section IV-D) to reduce the performance impact of this, before evaluating the impact on the cache coherency system using multithreaded shared memory benchmarks.

1) Single-Threaded Benchmarks: Unless we can be sure that a program isn't reading from or writing to data from other processes running concurrently on other cores, or the process isn't locked to a particular core, then we still need to use the techniques presented in section IV to prevent error propagation to separately checked code, even for single-threaded workloads.

Figure 8 presents the same single-threaded benchmarks as figure 6, but with additional bars for timestamps and data blocking in the L1 cache (L1 Timestamps) necessary for preventing error propagation, as presented in section IV-A, along with the variable length checkpointing (L1 AIMD Timestamps) presented in section IV-D used to reduce the overheads in cases of conflict misses causing the triggering of early checkpoints and paused main core execution.

Three benchmarks are particularly impacted by blocking data from being evicted from the L1 based on timestamp data. The first of these, randacc, suffers because of its highly random memory-access pattern, with little temporal or spatial locality, causing a large number of both conflict and capacity evictions. However, since this results in the program being extremely memory bound even without error detection or correction, we can eliminate the overheads entirely by dynamically setting checkpoint lengths with AIMD timestamps. The overhead of additional checkpointing is negligible as the workload is not compute bound, and by shrinking the checkpoint length we have fewer stores concurrently buffered in the L1 cache.

However, the performance for the other two benchmarks affected, freqmine and swaptions, is less optimal. Freqmine is particularly impacted, increasing overhead to 14% with L1 timestamps and reducing to 10% with variable length checkpointing. While both workloads suffer from frequent conflict evictions, as with randacc, the relevant data is typically temporally local, and available in the L2 cache, unlike with randacc, and thus the code is not as memory bound. Reducing the size of checkpoints is less effective for conflict than capacity misses, therefore variable-length checkpointing does not entirely solve the problem of overheads from using the low-associativity L1 cache as a buffer. A higher-associativity cache or victim buffer could mitigate this considerably.

Figure 9 shows the delays observed once we add in AIMD variable timestamps. Again, we see that typical delays are reduced further with respect to both schemes in figure 7. This is particularly true for randacc because the checkpoints are smaller, so average delays are reduced considerably.

Indeed, in figure 10 we see that, while most benchmarks spend over 90% of their execution time with a maximal instruction window of 5,000 instructions, randacc, freqmine and swaptions spend the majority of their execution with much smaller checkpoint lengths. Freqmine spends only 21% of its time at this maximum, while randacc spends 2.4% of its

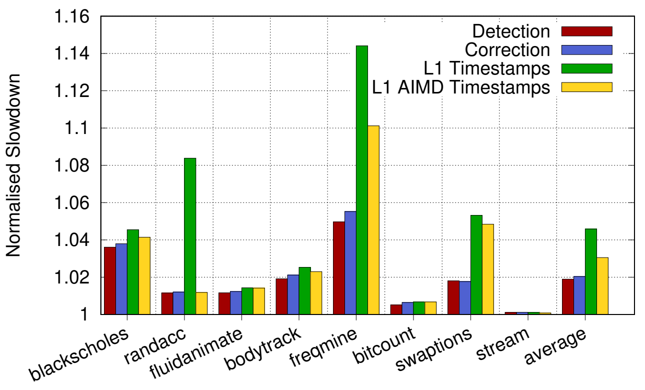


Fig. 8: Normalised slowdown for each single core benchmark, with the schemes from figure 6, along with the L1 timestamp scheme necessary to ensure correctness in multicore shared memory environments, and the AIMD timestamp length scheme to improve its performance.

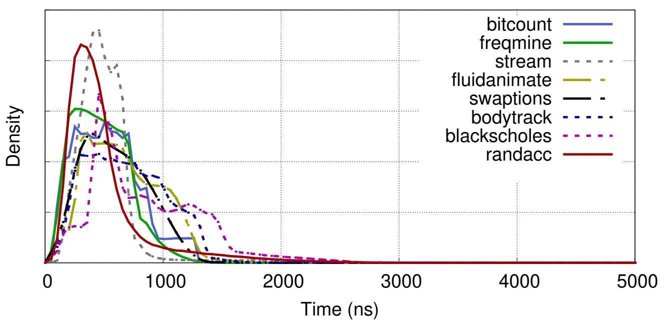


Fig. 9: Density plot to show the delays observed between a store committing and being checked, with data being buffered in the L1 and a dynamic AIMD timestamp length scheme.

time there, and swaptions less than a thousandth of a percent. The capacity and associativity of the L1 are too low for maximal performance on these benchmarks when the L1 is used as a buffer for checked data, with a fixed checkpoint size. However, performance is still relatively high provided we alter the checkpoint size dynamically.

2) Multithreaded Benchmarks: While the single-threaded benchmarks may suffer from using the L1 cache as a buffer for unchecked results due to cache evictions, with true multithreaded shared-memory workloads the additional problem of shared data emerges. As we discuss in section IV-C, other cores may force data to be flushed from an L1 cache, or be directly shared with another core.

Figure 11 shows the performance of Parsec [13] benchmarks running with two threads with our schemes and detection alone. With the addition of timestamps in the L1 cache, to prevent inter-thread communication before error checking, we do observe some slowdown. However, this is surprisingly slight, and almost entirely mitigated with the use of AIMD timestamps, to vary checkpoint lengths based on the amount of communication between cores.

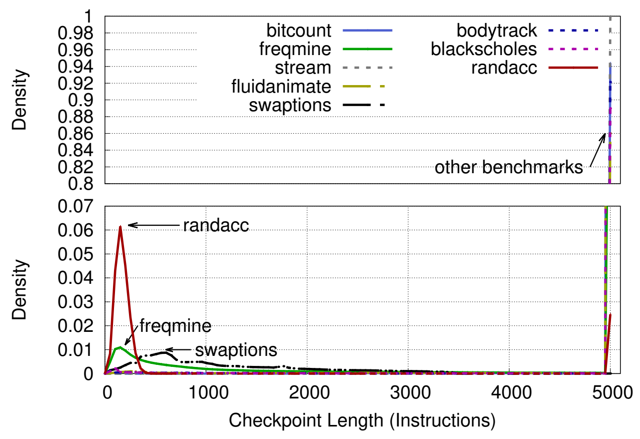


Fig. 10: Density plot of timestamp lengths during execution when using variable length AIMD timestamps (section IV-D).

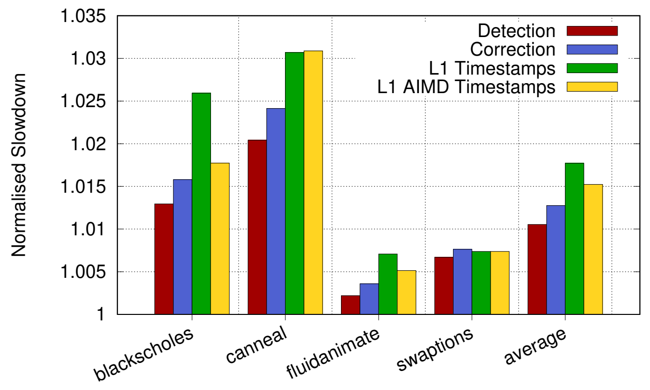


Fig. 11: Normalised slowdown for Parsec workloads running on two threads, with techniques from figure 6, along with the L1 timestamp scheme necessary for correctness in multicore shared memory environments, and the AIMD timestamp length scheme to improve its performance.

Still, in many ways this should be expected. The Parsec benchmarks are designed to scale well to multiple threads, and even on current systems, frequent communication causes programs to scale poorly. Because of this, we can say more generally that, for programs with high thread-level parallelism, the additional performance loss from delaying communication before error checking, to prevent error propagation and therefore allowing recovery from errors, is slight when compared with just providing error detection.

C. Other Overheads

Our prior work [8] places other overheads at 16% and 24% for power and area respectively for their heterogeneous detection system. We should expect ParaMedic to be similar: the additional overheads we are L1 data cache tag timestamps (a fraction of a percent of core area), along with the hard error table, AIMD unit and commit order tracker, which are small units that are insignificant in overall area and power

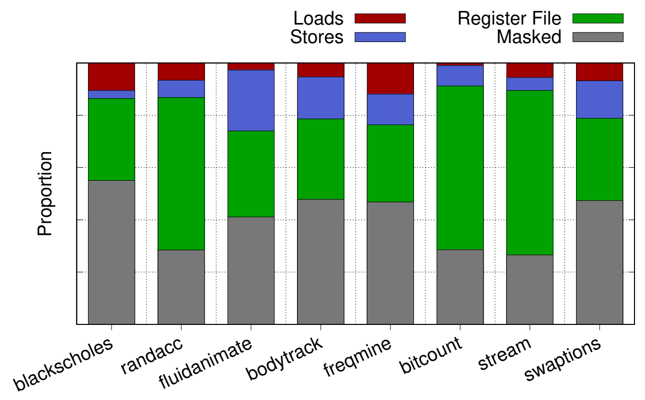


Fig. 12: Graph showing how errors injected into the register file propagate through the system.

calculations. While the register checkpoints store more state than in our prior work [8], we have implemented this in an area-neutral way by reducing numbers of entries stored, so this is taken into account in our performance numbers.

D. Fault Injection Study

ParaMedic re-executes all computation by the main core on checker cores, and so captures all observable errors that dual-core lockstep would: the difference is that computation may be checked in a different order from the dual-core lockstep execution due to the exploitation of parallelism.

However, the way errors manifest may differ due to the register checks at the end of each segment: in dual-core lockstep, these errors would instead appear the retirement of an instruction, and in other sequential error detection mechanisms [15], [4], [16], would either affect a future load or store, or be masked entirely. By comparison, in ParaMedic we must also check register files to ensure continuity between parallel checked segments. We explore this in figure 12. Here, we inject a random bit error in a random integer register at the start of each checkpoint, to observe the error's propagation. We see that errors are split significantly between load addresses, store addresses and data, and register files. We also see that up to half of all injected errors on the register file never affect the resulting computation: they are in registers that are never used by the application, and are overwritten by the time a checkpoint is completed.

E. Summary

For single-threaded code without shared memory, the extra slowdown from error correction is minimal, at just 2.0% compared with 1.9% for error detection. When we introduce shared memory, mandating communication limitations that prevent the escape of data from the L1 before it is committed by using timestamps, this increases to 4.6%. However, it can be reduced to 3.1% by using an additive-increase-multiplicative-decrease (AIMD) scheme to dynamically scale the size of timestamps, to reduce cache capacity evictions from data that hasn't yet been checked. When we extend this to multithreaded code, the pattern is similar: overheads of 1% for error detection increase to 1.5% for correction.

VII. Comparison with Transactional Memory

ParaMedic rolls back speculative execution in the presence of errors. There is, therefore, a clear comparison to be made with transactional-memory systems [12]. Further, we can consider the problem of error propagation as being akin to conflict detection, both within-core to subsequent checkpoints, and out-of-core when errors may propagate due to shared memory.

However, there are clear and necessary implementation differences between our technique and typical hardware transactional-memory (HTM) schemes. Most real-world HTM implementations are best-effort [12], meaning forward progress isn't guaranteed. This is possible because the system can always fall back on sequential execution of transactions, whereas there is no alternative to checking the code for correctness when performing error correction. Whereas concurrent writes in transactions are a cause for rollback, for error checking we should expect addresses to be written to multiple times in different checkpoints. This means that buffering in the L1 is suitable for best effort hardware transactional memory systems [29], but is not suitable for error correction; intermediate writes need to be buffered as well for more fine-grained rollback and commit. Another difference is that we can partition checkpoints as we wish for error checking, whereas they are necessarily in fixed points for atomic transactions.

Our error correction solution is, in transactional-memory terms, an eager versioning, optimistic conflict-detecting system up until the L1, and a lazy versioning, pessimistic conflict-detecting system between cores [30]. Optimistic "conflict detection" between checkpoints on the same core is necessary to achieve the desired parallelism: if we were pessimistic, any data accessed in multiple checkpoints would cause a stall until the previous check had completed. Similarly, eager versioning, where we use an undo log instead of using the L1 cache as a buffer of data which can be thrown away, is necessary to avoid the fates of all concurrent checks being tied together: we need to be able to overwrite uncommitted data in the L1 for future transactions without stalling, and also to be able to commit data within the L1 without forcing write-through to the L2. Without an undo log, intermediate rollback states become inaccessible. This would result in stalls while waiting for the last in a set of checks to complete, reducing performance. As all code is within a fault tolerance "transaction", this means the amount of sharing involved leaves lazy versioning impractical from a performance perspective, unlike in typical hardware transactional-memory implementations [12].

However, we disallow uncommitted data to leave the L1, and prevent the sharing of uncommitted data between cores. This, in effect, makes the transaction policy between cores somewhat like a lazy versioning, pessimistic conflict-detection system. We use lazy versioning in that the L2 is guaranteed to contain correct data, and the L1 is used as a write buffer. This makes rollback easier, as we don't have to serialise undo writes between different cores. For the same reason, we have pessimistic conflict detection, in that uncommitted shared data between cores is prevented by design, again to make rollback simpler by avoiding error propagation. A more optimistic solution might let potential errors propagate, and detect this at commit time, but this would be significantly more complicated from a verification and protocol perspective.

VIII. Related Work

A wide design space exists for providing processor fault tolerance: we compare the main categories here.

A. Hardware Redundancy

The two broad categories of hardware-only redundancy for computation can be described as space-based and time-based. Space-based schemes, such as lockstepping, duplicate hardware for redundancy. Lockstepping is used in ARM's Cortex-R series [14], which are designed for high reliability applications. Error correction can be achieved with lockstepping by using three cores with a majority voting system [6]. Lockstepping has also been used in other commercial designs such as the IBM G5 [31] and Compaq Himalaya [32]. Traditional lockstepping calculates the redundant results immediately, meaning fault propagation is trivial to prevent. However, it is often desirable to have the second core trail the first, as exemplified by Gupta et al. [33], to reduce cache misses and correlated errors. This has the side effect of increasing communication delay between multiple cores, as with our scheme.

Time-based schemes, by comparison, run the same code on the same hardware at different times, typically with hardware support to forward loads and stores between two threads. AR-SMT [15] is an example of a time-based redundant-multi-threading scheme without this forwarding, where the address space is duplicated instead to achieve the same effect at high overhead, whereas Reinhardt and Mukherjee [4] extend it with a load forwarder similar to that used by heterogeneous parallel error detection [8]. The overheads are still high, however: Mukherjee et al. [16] estimate a 32% performance overhead, along with the loss of a hardware context. Vijaykumar et al. [19] use tighter coupling between the two threads to ensure the checker thread executes before the main thread commits, to provide correction as well as detection. Time-based techniques tend to delay the checking of errors when compared with lockstepping, which finds them immediately, and so techniques such as SafetyNet [34] are necessary to roll back to consistent states. With ParaMedic, we can reuse many of the architectural elements from heterogeneous error detection [8], reducing the hardware necessary for arbitrary error recovery, allowing us to match the granularity of checking and recovery precisely, so we can free state as soon as it is no longer needed. Further, we can directly trap errors within a core's cache, preventing their escape to other parts of the system, or other systems, and guarantee forward progress even in the presence of hard errors, by tracking their behaviour within error detection.

B. Software Redundancy

It is also possible to achieve redundancy without any hardware support, by re-executing code and comparing the results in software, albeit at a significant performance penalty. SWIFT [35] is a solution which runs two copies of each instruction within a single thread to compare the results. Khudia and Mahlke [36] extend this by only repeating computation for error-intolerant parts of an application. Wang et al. [37] run the second execution in a separate thread, to make better use of multicore and multithreaded systems. Mitropoulou et al. [38] extend this by using a more efficient queue to share results between cores. However, for performance reasons hardware support is highly beneficial [4], [15], [16]. Hybrid schemes such as CRAFT [17] have also been proposed, which use compiler assistance to duplicate instructions, coupled with a special hardware detection structure.

C. Heterogeneous Redundancy

Our prior work [8] presented a heterogeneous parallel error detection scheme. This work focuses on exploiting the parallelism inherent in fault detection to reduce the power-performance-area (PPA) overheads of error checking, by performing the second run of a program on an array of smaller cores. Other work to exploit heterogeneity in error detection includes Austin's DIVA [39], which uses a superscalar in-order core to verify correctness of the execution of a larger out-of-order superscalar core. Errors on the in-order core are left unchecked through the assumption that it is implemented with larger transistors that are less susceptible to errors. However, extensive alterations to the main core's microarchitecture are necessary, including ECC on all register state.

Ansari et al. [40] utilise heterogeneity by pairing an older and newer version of the same microarchitecture series on a chip. If the newer core suffers from hard faults, it can be repurposed to provide hints for the slower previous generation core. By comparison, LaFrieda et al. [41] design for heterogeneity induced by manufacturing variability. They couple cores dynamically for lockstep execution, based on profiling, to maximise system performance by pairing similar cores.

IX. Conclusion

We have designed ParaMedic, an architecture for exploiting parallelism for error correction. This involves coupling a hardware undo log for rolling back errors, with using the L1 cache as a buffer for forwarding of unchecked values to future computation, without allowing it to escape to other cores to restrict the sphere of replication and thus avoid ordering problems upon rollback of writes within the load-store log. The system also allows recovery from hard faults, by detecting when repeated errors occur, and thus triggering hardware migration.

Performance is reduced relative to detection alone, but typically this is very minor. With an adaptive technique to set checkpoint lengths, the overheads increase from just 1.9% with detection, to 3.1% with correction. We therefore have provided a practical architecture to allow full error correction to be implemented extremely efficiently for commodity out-of-order superscalar systems.

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.37963.

References

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

[2] 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.

[3] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, 2011.

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

[5] 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, 2015.

[6] 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.

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

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

[9] http://www.anandtech.com/show/8542/cortexm7-launches-embedded-iot-and-wearables/2.

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

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

[12] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8," in *ISCA*, 2015.

[13] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[14] 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, 2015.

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

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

[17] 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.

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

[19] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *ISCA*, 2002.

[20] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[21] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's L1 data cache," in *ICCD*, 2006.

[22] ARM Ltd., http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500g/CHDEEHDD.html.

[23] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, 1989.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA*, 1995.

[25] 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.

[26] 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.

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, 2011.

[28] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *ISPASS*, 2014.

[29] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *SC*, 2013.

[30] A. Mcdonald, "Architectures for transactional memory," Ph.D. dissertation, 2009.

[31] 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, 1999.

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

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

[34] 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.

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

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

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

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

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

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

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