ParaDox: Eliminating Voltage Margins via Heterogeneous Fault Tolerance

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

Lionel Zoubritzky
École Normale Supérieure – PSL
Paris, France
lionel.zoubritzky@ens.psl.eu

Alan Mycroft
University of Cambridge
Cambridge, UK
alan.mycroft@cl.cam.ac.uk

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

***Abstract*—Providing reliability is becoming a challenge for chip manufacturers, faced with simultaneously trying to improve miniaturization, performance and energy efficiency. This leads to very large margins on voltage and frequency, designed to avoid errors even in the worst case, along with significant hardware expenditure on eliminating voltage spikes and other forms of transient error, causing considerable inefficiency in power consumption and performance.**

We flip traditional ideas about reliability and performance around, by exploring the use of error resilience for power and performance gains. ParaMedic is a recent architecture that provides a solution for reliability with low overheads via automatic hardware error recovery. It works by splitting up checking onto many small cores in a heterogeneous multicore system with hardware logging support. However, its design is based on the idea that errors are exceptional. We transform ParaMedic into ParaDox, which shows high performance in both error-intensive and scarce-error scenarios, thus allowing correct execution even when undervolted and overclocked. Evaluation within error-intensive simulation environments confirms the error resilience of ParaDox and the low associated recovery cost. We estimate that compared to a non-resilient system with margins, ParaDox can reduce energy-delay product by 15% through undervolting, while completely recovering from any induced errors.

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

I. Introduction

As microarchitectures evolve under the triple constraints of reducing the size of processors and their power consumption while increasing their performance, hardware errors grow more common [14], [19]. The causes of those faults are numerous: cosmic radiation, voltage fluctuation, defects in the die, and many others [63], and each comes with different effects. Although many errors may be naturally tolerated by the system—their effects being masked and thus not propagating to program output—a single one may result in an overall failure that necessitates starting over. Worse, some errors might remain unnoticed while corrupting the output or behavior of the program (and so result in silent data corruption). The associated cost can be particularly high [26], [47], and further improving performance incurs the risk of additional errors.

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/P020011/1, and the Trinity College / École Normale Supérieure Exchange Scheme. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.61808.

While conventional commodity processors do not feature redundancy at the hardware level, manufacturers already sacrifice performance for reliability, by using costly safeguards to reduce error likelihoods [70], on both voltage and frequency margins, and to mitigate voltage spikes [32], [57]. Voltage margins are used to decrease the probability that a transistor exhibits errors when faced with an electrical fluctuation; undervolting can be used to claw back energy lost to these margins [51], [65]. Conversely, CPUs can be configured to run faster than their specification allows, a practice known as overclocking [54], but this incurs the risk of timing errors that voltage and frequency margins are used to prevent.

Recent advancements in fully redundant execution allow full error detection and correction at very low cost. Ainsworth and Jones' ParaMedic [8], [10], which has generated industrial IP [9], is an architecture that uses the fact that duplicate fault-tolerance runs of an application are more amenable to parallelization than the original computation. This allows the execution of fault-checking code to be split up onto many small, highly efficient cores, reducing overheads by an order of magnitude compared to dual-core lockstep. When full fault tolerance can be achieved at lower overheads than are already paid by overvolting and underclocking commodity systems for resilience [54], this raises the question of what happens if we eliminate these margins, and instead achieve correct execution by utilizing redundant hardware to check and repeat incorrect computation. This perspective gives the potential for both better error coverage and lower power consumption even for commodity processors. Alternatively, as all such designs are energy-limited, it gives the ability to increase performance by overclocking, without affecting correctness.

Ainsworth and Jones' model [10] assumed that the time taken to handle actual errors is negligible. While this is consistent with the low observation rate of errors at large [33], this paper aims to use such a technique in deliberately error-intensive scenarios by lowering guard bands. In these situations, performance overheads from rollback can become significant. We design a new system, ParaDox, with adaptive mechanisms to reuse this parallelism strategy in error-intensive environments. ParaDox, with its dynamic error adaptation and efficient rollback, increases overheads only marginally compared with error-free fault tolerance. We present:

- A new method to simultaneously improve performance

and reliability compared with commodity systems, by combining efficient hardware fault tolerance with aggressive, error-seeking voltage thresholding.

- Techniques to minimize ParaMedic's performance loss at high error rates, by dynamically adjusting checkpoint lengths and redesigning structures to optimize rollback, dynamic adjustment of error rates through frequency-voltage feedback mechanisms, and rescheduling of checker cores to enhance power-gating potential.
- An error-injection simulator for ParaDox, including a variety of fault models, to observe the extent to which, under deliberately induced heightened error rates, ParaMedic loses performance that ParaDox can regain.
- A comparison of simulated overheads for ParaDox versus undervolting data taken from real systems, providing the expected energy savings for such systems with the addition of fault-tolerance hardware.

ParaDox gives an estimated power reduction of 22% through undervolting, at a 4.5% typical slowdown, giving energy-delay-product reductions of 15%. Alternatively, a 15% reduction in power consumption can be achieved by restoring this performance overhead through dynamic overclocking.

II. Background

A. Errors in digital circuits

Microprocessors are made of circuits that may contain a variety of vulnerabilities, usually classified into two categories. First, hard faults cause errors that repeat over time, most often due to a physical defect [64]. Second, soft errors are single-event upsets, typically caused by cosmic radiation, electrical noise [16], and voltage fluctuation [63]. In the latter case, an electrical component (e.g., transistor or capacitor) manifests one or more soft errors by immediately yielding an incorrect state, which may propagate. However, the component does not remain affected and resumes its normal behavior afterwards.

While the individual transistor soft-error rate tends to decrease with size because of the reduced likelihood of being hit by a cosmic ray [56], the overall contribution of radiation towards soft-error rates increases when components shrink because the total exposed surface does not vary much between technology generations and any electrical fluctuation generated by radiation has more effect on smaller components [63]. Higher temperatures combined with smaller transistors also increase variance and the risk of soft errors [20]. Hence, manufacturers incorporate margins in voltage and frequency to minimize the soft-error rate [36].

Errors in general appear either in combinational logic (e.g., register latches, functional units) or in memory. Handling the latter case has been extensively studied because of the early predominance of faults in this area, and error-correction code (ECC) has been widely adopted as a practical means of minimizing its impact. Combinational vulnerabilities, on the other hand, are by nature dynamic, hence they cannot be checked statically, like ECC, since the computation itself needs to be duplicated in some way to provide resilience.

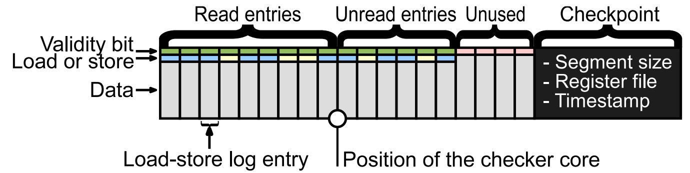


Fig. 1: Structure of a load-store-log segment.

B. ParaMedic

ParaMedic [8], [10] is an architecture that leverages highly parallel execution checking to provide error resilience with low performance and power overheads. The architecture uses two different kinds of processor: out-of-order superscalar main cores and simpler checker cores. Each main core is assigned a large number of small, slow checker cores. The main cores are commodity processors used in everyday computing environments, having a medium-to-large out-of-order pipeline, and perform the initial computation. The checker cores work in-order at a low frequency; they are also several orders of magnitude more power and area efficient [1], [2], [3].

The computation of a main core is divided into segments at run-time, consisting of a series of contiguous committed instructions. At the end of each segment, a checkpoint is taken, which keeps track of the current architectural state of the main core and the number of committed instructions for the segment. Upon checkpointing, a checker core is launched with the architectural state of the previous checkpoint and is expected to redo all the computation of the past segment until it reaches the same number of committed instructions. An error is detected if there is a discrepancy between the final architectural state of the checker and the one that has just been recorded. Checkers do not actually have access to main memory on the data side: their data cache is replaced by a load-store log that records the memory operations performed by the main core during the appropriate segment. Each time a main core executes a load from memory, the loaded value is kept along with its address in the load-store log. Similarly, each store is logged along with its address and the previous value at that address. Each time a checker performs a load, it reads the value kept in the load-store log, and whenever it attempts a store, the new value is compared to the one in the log, an error being detected upon mismatch. The main and checker cores have different paths to the cache system that do not share any logic; the latter reads values from the load-store log that is written to by the load-forwarding unit [8].

The log is itself divided into segments, as illustrated in figure 1, which correspond to the run-time segments of the main core. Since the order of the operations is kept identical between the main and the checker cores, each segment of the load-store log acts as a queue, which accelerates memory operations for the checker cores compared to a normal cache.

Different checker cores check different segments of the computation of their main core simultaneously. The underlying fault-tolerant architecture therefore executes every instruction twice, and duplicates loaded values at the cache, reading them

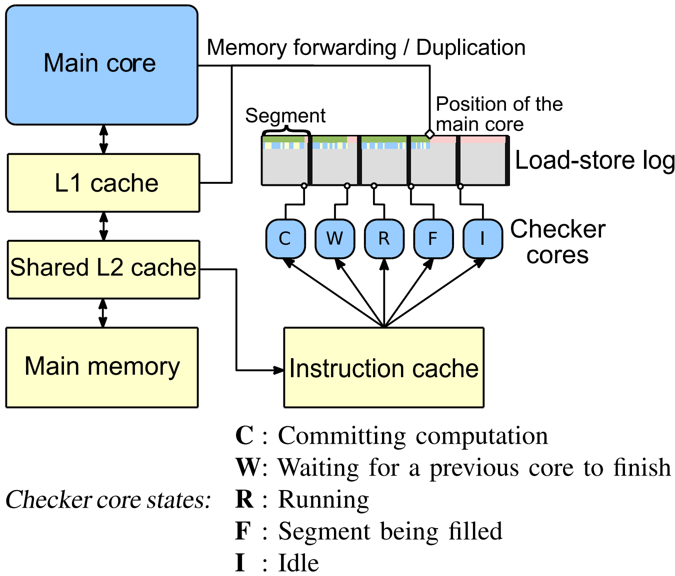


Fig. 2: Heterogeneous error checking in ParaMedic.

out along a different path, thus any error in the committed instruction stream (or propagated to memory) is handled, and cannot propagate system-wide. Any full lockup of a core is detected via timeout. If all checkers are busy when a main core reaches a checkpoint, the main core has to wait for a checker to finish, introducing further slowdown in addition to the cost of checkpointing itself. Since we may have multiple main cores working in parallel and possibly communicating, unchecked values are buffered in the L1 cache until checks are complete, with dynamic mechanisms to trade off communication and checkpoint frequency. Stores that are uncacheable must be checked before they can proceed. The overheads of this are managed by dynamically adjusting checkpoint lengths based on memory-mapped-access frequency. Syscalls are considered as standard operations that can be rolled back, unless they update external state.

The starting architectural state, and the stores belonging to the segment of a checker that is waiting for older checker cores to finish, are kept in the log as long as the checker is waiting, in case an error occurs in those older checkers. Upon error detection, the main core stops its execution, all the stores that happened between the beginning of the faulty segment and the current state—which are all kept in the load-store log—are reverted, and the main core is reset to the architectural state kept as the starting point of the checker that discovered the error. The five possible states of a checker core and the design of ParaMedic are summarized in figure 2.

The correctness of the system comes from the principle of strong induction: if all segments up to one point have been verified independently, then the computation up to that point is correct; the rest of the computation up to the current state is still under verification and can be entirely reverted. ParaMedic offers comprehensive coverage since every committed instruction is re-executed on a different physical substrate. Yet the performance overhead is minimal and the additional hardware complexity is much smaller than what would be needed for triple-core lockstep [35], thanks to the use of heterogeneity, and since all forwarding and decoding decisions are duplicated, no ECC is required within the main core.

III. Motivation

Counterintuitively, it is actually a constraint to assume that errors should be exceptional. Indeed, once a fault-tolerant design has been conceived, the common case assumption should be that errors can occur (and will be corrected). The opposite to this hypothesis, required by non-resilient designs, imposes penalties in terms of power consumption and clock frequency [32], [51], [57], [65]. However, allowing errors in the general case is not straightforward either and requires assessing the risks and the potential benefits precisely. In particular, an architecture designed to recover from a few sparse errors may not be able to handle higher error rates. Hence, such a design needs to be tested against actual errors, both to ensure its validity for the single-fault case and then for the general one.

In this section, we explain how to leverage error resilience to improve the performance and energy consumption of a system through undervolting and overclocking. We motivate the need for an experimental analysis of the behavior of a fault-tolerant design faced with actual errors, to give quantitative measurements of error-recovery time, and the limitations the existing ParaMedic [8], [10] design faces.

A. Margins for error-free computation

On non-resilient systems, errors must be avoided at all costs to guarantee correctness and to prevent escalation to a potentially catastrophic failure. Hence, manufacturers place margins on the voltage and clock frequency of their products, to ensure that even a slight fluctuation in the power input, or a single cosmic ray, does not alter any part of the computation [70]. Numerous studies have focused on inherent software resilience to hardware errors [43], or proposed software mitigation techniques to reduce or remove the effect of a fault. Previous work has designed systems that can give an indication of voltage errors starting to appear, by using representative regions with ECC [12], [13], or monitors to detect some forms of timing violation and noise spikes [24], [66], without the ability to handle metastability [15]. We argue that allowing widespread errors to happen on a more regular basis, only to be caught and corrected by a full fault-tolerant architecture, deserves consideration. Instead of reducing reliability only to catch a subset of the resulting effects, such an architecture could improve reliability relative to a standard-margined baseline, while mitigating and even inverting the resulting power consumption overhead via undervolting techniques and the removal of margins and spike mitigations.

Undervolting reduces the power consumption of the machine, but incurs the risk of timing errors because of the reduced energy gap between the two states of a transistor. Overclocking, by comparison, makes a CPU run at a higher clock frequency than its nominal one. Figure 3 shows, schematically, the evolution of the total power consumption of a fault-tolerant chip

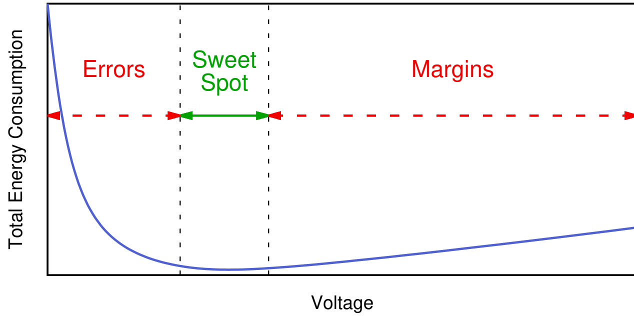


Fig. 3: Error-resilient undervolting in theory.

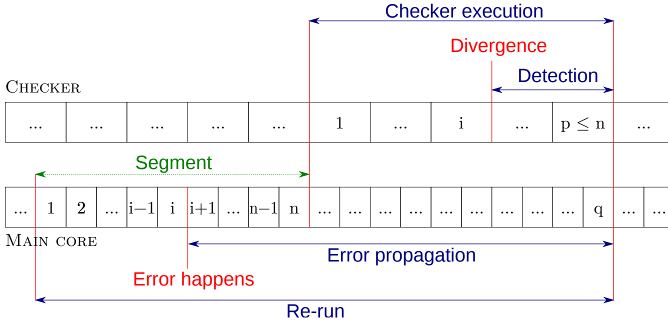


Fig. 4: Timing breakdown for a single error in ParaMedic.

with the voltage level. On the left, too much undervolting causes errors to appear, and recovery requires additional energy. On the right, with no undervolting, excess power is needed because of the voltage margins. Minimal energy consumption is attained between these two areas. Overclocking and undervolting could thus be aggressively used to optimize the performance and energy consumption of any fault-tolerant chip. However, this requires adequate fault tolerance: if errors are to become commonplace, detection and recovery costs must be low enough to avoid dominating execution time. This could be problematic because error-resilient designs are not usually tested against actual errors, so the costs involved when facing them are at best theoretical, and typically not mentioned.

B. Error-recovery costs

Figure 4 summarizes the timing of an occurence of an error, then its subsequent detection and recovery in ParaMedic. The lower band corresponds to execution of the main core and each tile corresponds to a single instruction; the upper band corresponds to the checker running for the considered segment. In the figure, an error happens at the ith instruction of the segment, which is n instructions long. After the last instruction, the checker core is launched. The length of each tile is longer for the checker, to represent it being slower to execute instructions. The divergence in behavior of the checker and the main core appears when the checker executes the ith instruction, but there may be additional delay before it is detected as a change in state, bounded by the segment size.

Upon detection, the main core rolls back all the memory writes that happened since it executed instruction 1 of the segment, then it re-runs everything from that point. The overhead from recovery is thus comparable to the "Re-run" part of the figure, plus the rollback cost, which is a linear function of the "Re-run" length. This is wasted computation, exacerbated by the assumption that errors are rare, and thus that we can delay their checking for a long time and roll back any subsequent computation. Under a scenario where we deliberately increase the frequency of errors, this design point is suboptimal and we instead need to design a system that performs well with both a high and low frequency of errors.

IV. PARADOX

To make ParaMedic suitable for use in deliberately error-intensive scenarios, we extend it to ParaDox. ParaDox is a system able to deal with both high and low error rates with good performance, and able to dynamically control voltage and frequency to influence error rates and energy consumption. We introduce new optimizations to improve ParaDox's power-gating potential, reducing its overall energy-consumption overhead and thus improving its capability to achieve better performance than commodity systems (given their current voltage and frequency margins), and to reduce the cost of rolling back checkpoints in the presence of errors.

A. High-error adaptation

ParaMedic assumes that errors are rare, and thus large amounts of computation can be checked at once, all of which can be rolled back if any part is found to be incorrect. This does not directly affect correctness, as incorrect execution will always eventually be rolled back. However, it will cause performance loss if errors are frequent because the large amount of execution subsequent to an error is wasted and must be reverted. Even worse, with very high error rates this can result in livelock. Still, with low error rates, frequent checkpointing lowers performance by causing many copies of the register file to be taken, blocking commit for 16 cycles [8].

ParaDox uses a dynamic approach, where we maximize performance by increasing checkpoint length when error frequency is low, and decreasing it when error frequency is high. ParaMedic already uses such an approach for communication between cores, and so we use the same technique, an additive-increase multiplicative-decrease (AIMD) scheme, for errors. If an error is observed in a checkpoint, we halve the target instruction window for the following checkpoint. If no error is observed, we increase the instruction window by 10 for the next checkpoint, up to a limit of 5,000 instructions; the maximum is set such that checkpointing cost is negligible, but the worst-case recovery time is still bounded by a relatively short stream of checked instructions, and the increment of 10 set to allow a steady increase under a phase change.

Still, the multiplicative decrease above is sometimes not rapid enough. When applications change phase quickly, this can cause slowdowns, where the assumed level of delay tolerance is too high, and thus the main core is kept stalled while the checker cores catch up. This is not just caused by high error rates; another example is when the L1 cache's buffering of unchecked, but written to, cache lines, necessary for a total-order rollback on multicore systems, is expended. Ideally, the total checkpoint

length of all of the checker cores combined should not cause an eviction attempt of an unchecked dirty cache line, as this eviction must wait until a check is complete. To make this adjustment more rapid, while avoiding instability in checkpoint length, we add a new factor in the calculation. On a checkpoint-length reduction (either from an observed error, or from an eviction attempt), ParaDox sets the new checkpoint length as being the minimum of half the current target length, and the actual observed length of the previous checkpoint, which may be smaller as a result of an eviction attempt, a discovered error part-way through the checkpoint, or reaching load-store-log capacity. This can allow ParaDox to outperform ParaMedic, in addition to its new ability to optimize performance in the presence of errors.

B. Dynamic voltage adaptation

Adapting the checkpoint length fits it to the error rate, but that error rate can increase if the voltage gets to a threshold that is too low. In this situation we also want to dynamically adjust the system, this time reducing the error rate by trading off power consumption to increase the voltage. We would like to use a similar AIMD mechanism to dynamically move between safe voltage levels and possibly unsafe voltages that have lower power consumption. We can do this by halving the difference between the current and known safe voltage (thus increasing supply voltage) on an error, and otherwise increasing the difference by lowering voltage.

However, there are two issues here. The first is that by halving the difference between current and safe voltages on each error, we will spend a significant amount of time using more power than is strictly necessary for safe operation. Instead we use a multiplicative factor of .875 [68] to balance the fast return-to-safe operation with the need to optimize a highly performance-critical variable. Still, we want to spend as much time at low voltages as possible; as ParaDox can recover from errors, we should typically operate below the point of first error. We record the highest voltage at which an error was seen so far, and use this to adapt the voltage decrease factor, which slows down by a factor of 8 below this tide mark. This causes ParaDox to spend more time in error-seeking regions before an error is observed. This tide-mark error point is reset every 100 errors, to allow ParaDox to become error-seeking again, in case the application or system has changed to a more voltage- or error-tolerant phase.

The second issue is that, while sudden significant voltage increases on an error will allow us to quickly move to an error-free guard band, sudden shifts may instead add voltage spikes into the system, which may cause further errors that will need to be rolled back. Instead, another way we can temporarily return to correct execution at the current voltage is by dropping frequency. To smooth out the voltage response while maintaining fast reaction to errors, and optimizing for performance, we instead use the AIMD-set voltage as a target, which a controller can dynamically move towards based on the timing limits of the regulator circuit. While the current

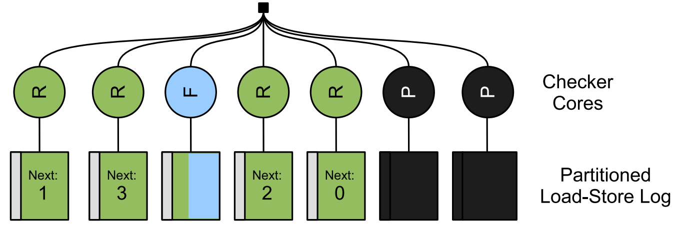


Fig. 5: ParaDox doesn't allocate checker cores round-robin. Instead, it allocates the lowest-indexed free checker core and log to execute and store the next checkpoint, allowing us to power gate the logs and cores of higher indices. Key: F = filling, R = running, P = power gated.

voltage is lower than the target, we scale clock frequency to compensate. This is based on the formula:

$$f_{\text{current}} = f_{\text{target}} \times \frac{v_{\text{current}} - v_{\text{threshold}}}{v_{\text{target}} - v_{\text{threshold}}}$$

where f is frequency and v is voltage, and we assume that attainable frequency is proportional to supply voltage minus threshold voltage [21].

There are other reasons for dynamically adjusting the level of sub-margin voltage we are willing to accept. Different workloads use different parts of the hardware, each of which will start to hit timing limitations, and thus exhibit errors, at different voltages. ParaDox allows the best performance to be dynamically tuned for any workload. For example, workloads not using the floating-point unit need not preserve timing for the FPU path. We assume that individual main cores have their own voltage islands, so each can change independently. Each group of checker cores will have a separate, common island, to allow them to run a significantly lower clock frequency than the main core, and without undervolting.

C. Aggressive checker gating

To avoid significant slowdown to a main core, the parallel checker cores collectively must go further than just matching the average instructions-per-cycle (IPC) of the core they are checking. Instead, the minimum IPC of the checker cores must match the maximum of the main core, otherwise certain instruction combinations will cause slowdown in some scenarios, reducing overall performance. For example, the divide unit of a checker core may be considerably lower performance than its other units, as a proportion of the main core's execution units. Long checkpoints, enabled by buffering many loads and stores per checkpoint in the load-store log, help to mitigate this by aggregating many instructions together. As slowdown is only incurred when all checker cores are busy, the aggregate effect on instruction throughput of a few slow instructions will be minimal, and can be hidden by those that are comparatively faster. Still, programs often sit in tight loops that perform the same computation repeatedly, meaning any workload that the checker cores suffer at will consistently lower performance. Checker cores must provide good enough performance even for the worst case.

However, this offers an opportunity since it implies that for the majority of the time, some checker-core resource will be underutilized. This means we should actively try to avoid powering checker cores when they are not needed. To optimize this, we schedule checking to favor using fewer checker cores if they are not all needed, rather than the round-robin strategy of ParaMedic. Instead, the next checker core (and thus log segment) to be scheduled is chosen as the free core with the lowest ID. This ID is then stored at the end of the log segment previously filled, to allow us continuity, and likewise at the front of the new log segment, to allow us to roll back stores in the event of an error. An example is shown in figure 5. To avoid uneven ageing, ID 0 is chosen at random at boot time.

This means that, under situations where fewer checker cores can keep up with the main core, we can entirely power gate unscheduled checker cores, rather than leaving them and their load-store-log segments powered and holding state.

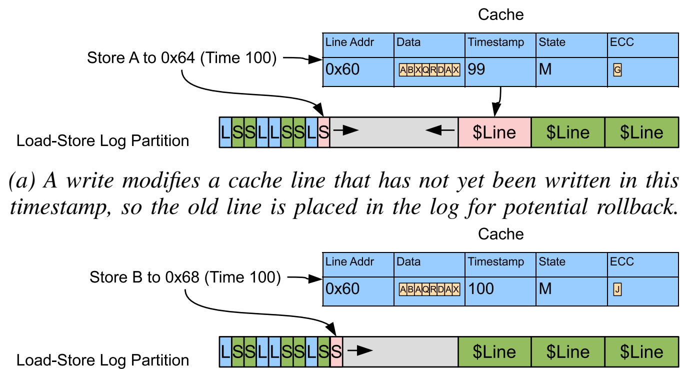
D. Line-granularity rollback

As ParaMedic is optimized for high performance in low-error environments, rolling back to a consistent state is a slow operation, performed by walking load-store-log segments in reverse to undo each store in turn. This uses similar state to the detection mechanism, in that for detection we need all words that have been loaded and stored, and for correction those stored words also store the old versions they wrote over.

However, under an assumption of temporal and spatial locality, this rollback is inefficient. While we could roll back to any uncommitted checkpoint we currently have stored, within a checkpoint we need only roll back to the earliest version of a write to a given location; all other writes will be overwritten in the rollback process. This means we need only store the first version of the write in each checkpoint in order to roll back to all consistent states possible to return to.

We can identify whether a cache line has been written to within a given log segment using existing state within ParaMedic. Each line in the L1 cache features a timestamp to prevent eviction of uncommitted data, which we can reuse to work out whether we need to store an old version of a cache line. If this timestamp is less than our own currently executing timestamp on the main core, we take a copy of the cache line, insert it at the top of our SRAM load-store-log segment, then set the timestamp in the cache as our own. Otherwise, we do not need to store the old version in the cache, and instead store only the newly written word in the detection segment.

Since this timestamp is stored per cache line to reduce overheads, we must also store our rollback data in cache-line format. Still, this simplifies metadata management by allowing us to copy all ECC from the cache line itself rather than recalculate any for the data word or address. Under the assumption that programs show temporal or spatial locality, this scheme is likely to use less space in the log than recording individual words multiple times. The cache line's physical address is stored in the log, to allow rollback without translation, whereas as with ParaMedic [10], loads and stores to be checked are stored with the virtual address to avoid translation on



(a) A write modifies a cache line that has not yet been written in this timestamp, so the old line is placed in the log for potential rollback.

(b) A subsequent write to the same line in the same checkpoint region has matching timestamps, so no copy of the cache data need be taken.

Fig. 6: ParaDox splits out rollback data at the cache-line granularity into load-store-log segments, optimizing rollback by storing only the oldest data copies within a checkpoint.

checker-core execution, with the original translation on the main core implemented redundantly.

This setup is shown in figure 6. Detection logs are stored on one side of the log; rollback cache lines are stored on the other. The current index for each is stored with the log, and once these two indices meet, or will meet following the commit of the next load or store, a new checkpoint is created and the current one issued to the relevant checker core.

E. Coverage

ParaDox is primarily designed to cover errors in undervolted or overclocked main cores through redundant execution on other hardware. This re-execution can cover any error inside the main core, as the execution is fully repeated, provided the same error does not also occur on checker cores. Further, assuming the checker cores are not undervolted or overclocked, the compute units of the system will be strictly more reliable than a non-redundant system with voltage margins, as even though the main core will exhibit errors, it is likely to be correct with high probability, therefore any errors on the checker cores (from, for example, cosmic rays) will be caught with high probability by not matching the main core execution, and will be rolled back and fixed. We could go further, and deliberately increase error rates on the checker cores through undervolting. As main and checker cores are microarchitecturally distinct, critical paths are unlikely to be in the same places, and thus errors caused by not meeting timing constraints are unlikely to result in common-mode errors across both main and checker cores. However, as the checker cores are already low energy, this is likely to result in significantly smaller savings than undervolting main cores, and its error properties are more complex to quantify, so we use traditional voltage margins on checker cores, leaving only errors caused by mechanisms such as cosmic rays.

ParaDox's redundancy only covers the compute elements of the chip since reliable systems usually cover memory using ECC bits [62], where we assume SECDED protection. ECC has been shown to be effective in protecting caches in the face

Main Cores	
Core	3-Wide, out-of-order, 3.2 GHz
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	32 KiB, 2-way, 1-cycle hit lat, 6 MSHRs
L1 DCache	32 KiB, 4-way, 2-cycle hit lat, 6 MSHRs
L2 Cache	1 MiB shared, 16-way, 12-cycle hit lat, 16 MSHRs, mostly incl, stride prefetcher
Memory	DDR3-1600 11-11-11-28 800 MHz
Checker Cores	
Cores	16× In-order, 4 stage pipeline, 1 GHz
Log Size	6 KiB per core, 5,000 inst. max length
Cache	8 KiB L0 ICache per core, 32 KiB shared L1

TABLE I: Core and memory experimental setup.

of voltage reduction [13], though in practice cache memories may be designed to meet all voltage and clock combinations at which ParaDox is used, with only compute logic undervolted.

V. Experimental Setup

To evaluate ParaDox's error resilience and performance under high-error scenarios, we model a high-performance system using the gem5 simulator [17] with the ARMv8 64-bit instruction set, and configuration given in table I. This is similar to systems validated in previous work [31] and the X-Gene 3 processor that has previously been used for undervolting experiments [51]. For overall performance metrics, we evaluate over SPEC CPU2006 [34], fast forwarding for 1 billion cycles then running for 1 billion cycles. For design-space exploration, we evaluate over compute-bound *bitcount* [30] and memory-bound *stream* [44] workloads used in previous work [8], [10]; these are intended to show the worst-case and best-case for the overheads caused by overly large checkpoints, respectively; the longer SPEC workloads would fit somewhere between these two extremes. We extend the multicore timestamp-based implementation of ParaMedic [10] to provide the adaptation, cache-line and power-gating techniques, along with a new fault-injection mechanism to test its performance under error-intensive scenarios. All baselines are relative to an unmodified, fault-intolerant system.

We do not test here on multicore workloads, because ParaDox does not directly change ParaMedic's behavior with respect to inter-thread communication. Still, the major cost of multicore ParaMedic, which is in buffering unchecked stores in the L1 cache [10], for single-threaded as well as multi-threaded workloads, is modeled, and improved upon, in the following evaluation.

A. Error-injection framework

We now present the error framework we use for analyzing the effect of faults on the performance of ParaDox. Figure 7 summarizes the key steps. The gem5 simulator enables fine-grained error injection that can target specific functionalities; various frameworks for error injection have been implemented on top of gem5 [55]. However, these are not compatible with ParaMedic's simulator because of the unique interplay between the main core and its checkers, whose microarchitectures differ. As a consequence, we have developed an in-house error-injection framework[1] for ParaDox and ParaMedic.

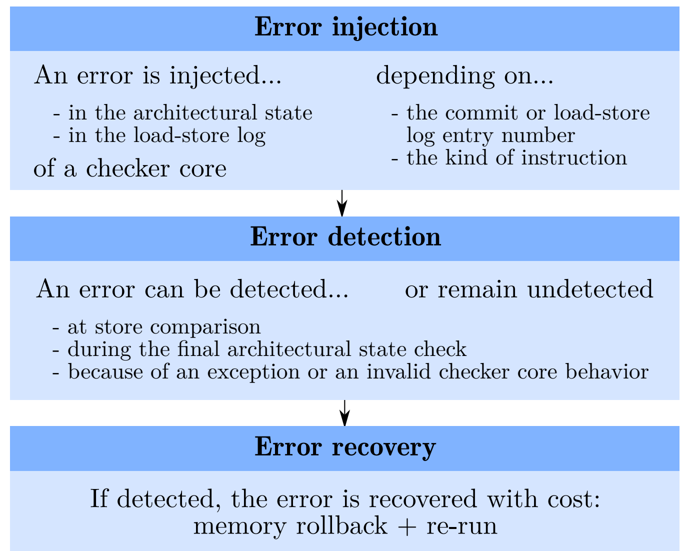


Fig. 7: Summary of error testing

In ParaDox, an error detected by a checker core can either indicate an error in the checker itself or in the main core. For the limited purpose of our hardware simulation, we choose to restrict error injection to the checker cores only. This simplifies error recovery without changing the results, as error detection is symmetrical; the mechanism is unable to distinguish which component caused the error, only that one is incorrect. Injecting into architectural state likely overestimates error frequency by a constant factor, but given the modeled error rate [65] is exponential, this cannot impact results significantly.

For the purpose of linking the error rate with the global slowdown of the system, we only simulate independent errors (as for cosmic rays). This is not entirely accurate since errors caused by undervolting may result in repeating patterns from not meeting timing constraints. However, since ParaDox changes frequency and voltage under such circumstances, duplicate errors are unlikely and thus a random injection suffices to measure recovery times. We thus choose the geometric probability distribution to govern the gap between two error injections. We inject errors in three ways, to approximate the wide variety of possible faults that can happen in hardware:

– Memory faults are represented by errors in the load-store log. In practice, this consists of simply flipping one bit of the data carried by a memory operation. The geometric gap between two injections corresponds to the number of targeted operations (either only loads or stores).

[1] This framework is available at https://doi.org/10.17863/CAM.61808.

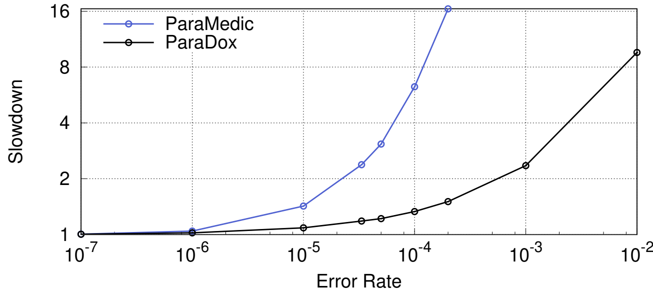


Fig. 8: Performance of bitcount under increasing error probabilities, relative to ParaMedic with fault-free execution.

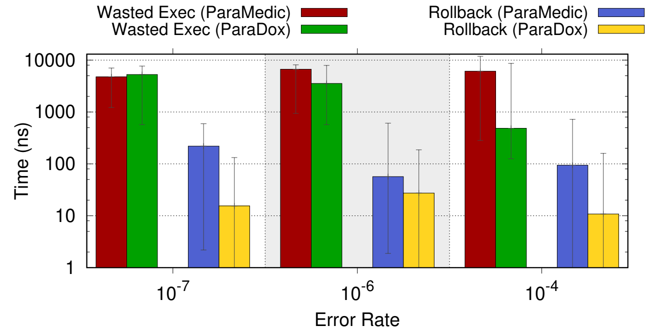
- Combinational faults from a defect in a particular functional unit provoke errors that only arise when specific instructions are executed. To tackle this category of unreliability, simulation compromises the targeted functional unit by corrupting the registers that have been modified by the concerned instructions. An instruction that has no effect is indistinguishable from a discarded instruction: no error is injected if no register is touched.
- Other combinational faults of unknown origin are simulated by flipping a single bit in a register, chosen at random among those of the targeted category (integers, floats, flags, or miscellaneous). The gap between two error injections on the architectural state is simply the number of instructions being executed.

Errors due to undervolting are generated using an exponential model following the formula from Tan et al. [65]. Its parameters correspond to the Intel Itanium II 9560 8-core processor with a nominal voltage of 1.1 V taken from this work, rather than the Arm system we simulate. This choice is driven by practical reasons, since no such study of the evolution of the error rate with the voltage exists for Arm processors; the closest related work would be the study by Parasyris et al. [54] of the minimal reliable voltage, but it does not indicate the evolution of the error rate past the first error. This setup is intended to match the rate of errors in an undervolted setup with a variety of different sources, rather than precisely matching the form of errors in undervolted systems, to capture performance effects.

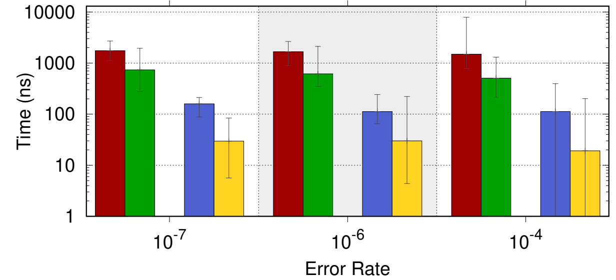
VI. Evaluation

A. Performance under high error rates

Figure 8 shows the performance of ParaMedic and ParaDox, relative to error-free execution under ParaMedic, at increasingly probable error rates. Both are suitable at error rates that are common in normal settings (even 10^{-7} gives approximately 300 errors per second, whereas a typical processor sees a negligible rate of fewer than one per year), but when we deliberately incur high error rates, ParaMedic quickly starts to suffer significant overheads. When 1 in every 5,000 instructions and memory operations incurs an error, ParaMedic slows down by 16×, as checkpoints are too long and copies of executions repeatedly incur errors, introducing livelock into the system. By comparison, ParaDox, with its dynamic checkpoint lengths that adapt to error rates, can achieve similar performance at error rates two orders of magnitude higher. In fact, 1 in every 100 operations (an unrealistic error rate in practice) must fault before a slowdown of 8× is reached.



(a) Bitcount



(b) Stream

Fig. 9: Comparison of the average overheads due to re-execution and memory rollback at low and high error rates. Error bars show range.

B. Analysis of recovery times

Error recovery comes in two parts: memory rollback first, then re-execution. In figure 9 we see how the average absolute recovery times differ between ParaMedic and ParaDox at various error rates; *bitcount* [30], a compute-bound workload, and *stream* [44], a memory-bound workload, are shown. For *bitcount* with low error rates, ParaMedic and ParaDox waste similar amounts of execution that is later found to be incorrect. However, at higher error rates, the average is brought down by an order of magnitude. This is because ParaDox adjusts its checkpoint lengths based on the observed error rate, and thus wastes less execution time. This is less pronounced for stream, which, due to being memory-bound, fills the load-store log quickly, and so has smaller checkpoints in general.

By comparison, rollback times for ParaDox are typically an order of magnitude lower regardless of error rate due to storing a cache line once per checkpoint, rather than each word every time it is written to, which allows for memory operations to be completed significantly more quickly. In general, the overheads from error recovery are dominated by wasted execution; while the ranges of re-execution and rollback cost overlap in some cases, wasted execution is typically between one and two orders of magnitude more significant. This is seen more strongly in the compute-bound *bitcount*, with its larger checkpoints, but is also seen to a lesser extent in *stream*. This is unsurprising, as ParaMedic and ParaDox are designed to tolerate high error-checking latency to facilitate their form of parallelism.

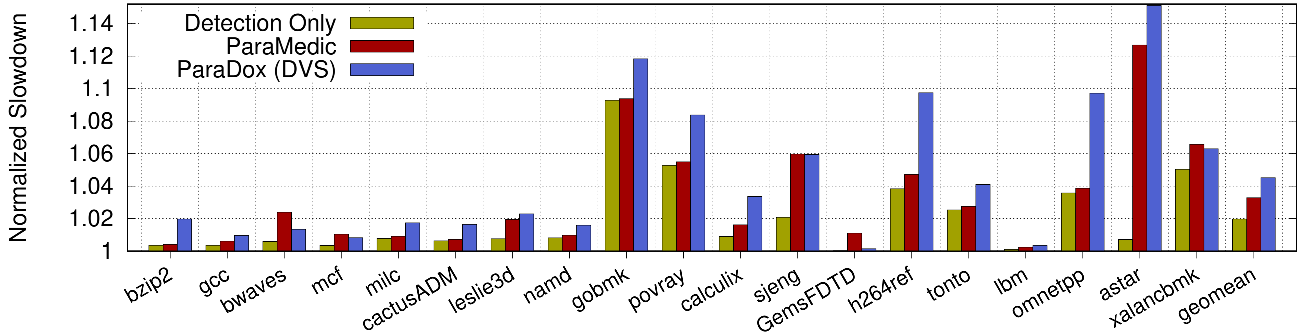


Fig. 10: Performance of ParaDox with dynamic voltage scaling on an exponential model (Tan et al. [65]), compared with error-free passive detection [8] and ParaMedic [10], all relative to a ParaDox-free baseline.

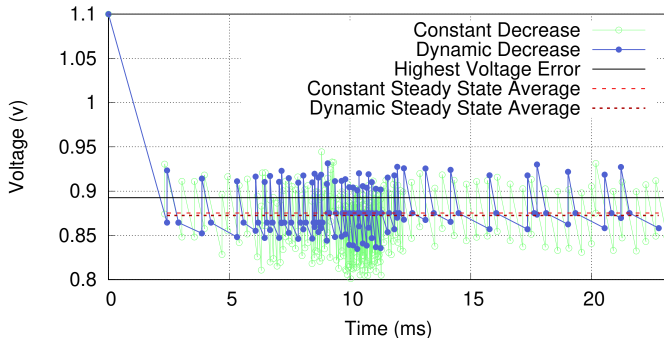


Fig. 11: Voltage over time on ParaDox running bitcount.

C. Dynamic voltage adaptation

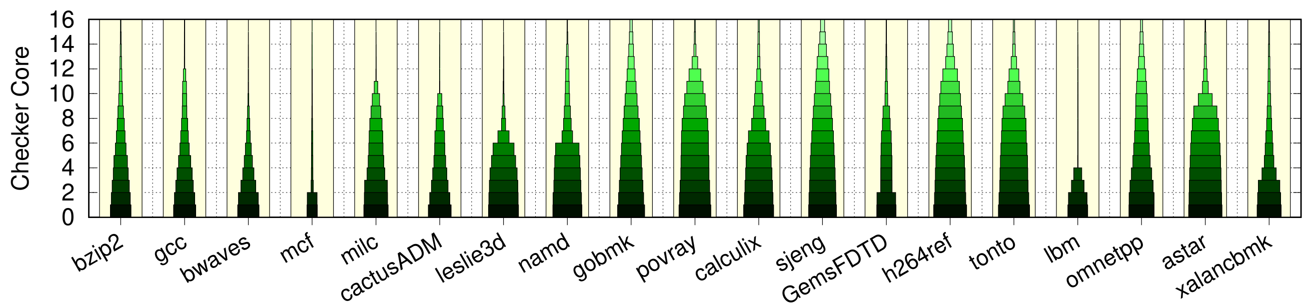
Figure 10 presents both error detection only [8] and ParaMedic [10] against the error-seeking voltage adaption scheme of ParaDox, with dynamic voltage scaling (DVS). Despite recovery overheads of up to 10 microseconds (figure 9), ParaDox rarely increases performance loss significantly. The main additional overheads, causing increasing slowdown from left to right, are 1) Register checkpointing and lack of checker compute (bar 1 in figure 10), plus correct handling of multicore data propagation (bar 2), plus rollback under frequent errors (bar 3).

Some workloads suffer from overhead as a result of heterogeneous error detection [8] alone. *Gobmk*, *povray*, *h264ref*, *omnetpp* and *xalancbmk* suffer from frequent misses in the checker cores' private instruction caches, which could be alleviated at the expense of more SRAM per checker core. Others, such as *milc* and *cactusADM*, suffer some overhead as a result of the checkpointing process, despite not being instruction- or checker-core-bound, which could be partially alleviated with a larger SRAM log. Other workloads (in particular *bwaves*, *sjeng* and *astar*) only suffer significant overheads once ParaMedic and ParaDox's rollback buffering techniques come into play, due to a combination of conflict misses affecting the amount of state that can be buffered in the L1, and lack of storage space in the partitioned load-store logs for old cache-line data. While some workloads (*bwaves*, *mcf* and *GemsFDTD*) overcome the induced errors and have higher performance than ParaMedic, due to the locality from line-granularity rollback (section IV-D) designed primarily to reduce rollback cost and the new checkpointing strategies, other workloads suffer either relatively minor performance loss from the rollback, or larger performance loss due to filling the logs earlier from a lack of locality.

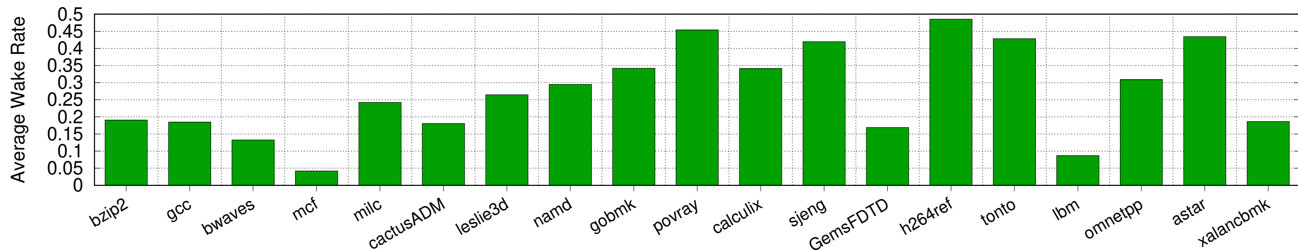
Figure 11 shows how voltage is scaled over time, both with ParaDox's default dynamic decrease, where voltage drop is slowed when below the recent highest-voltage error, and a static technique, which uses a constant voltage-drop rate. There are a number of observations we can take from this data. First, voltage increases, and thus rollback rates, are not constant over time. From between 5 and 13 ms, voltage decreases over time are more frequent than in other regions, as checkpoints are more frequent. This is because the checkpoints are smaller as the load-store log reaches capacity. Still, this behavior is desirable: when checkpoint sizes are already small, we can afford to be aggressive with voltage reduction, as less work must be rolled back on the discovery of an error. Second, the dynamic decrease mechanism produces far fewer errors than a constant decrease, despite achieving a lower average voltage and being equally responsive on the discovery of errors. Finally, both averages are significantly lower than the highest voltage observed error. ParaDox can perform well within voltages where errors are relatively frequent, and dynamically adjusts itself to act within these voltages.

D. Aggressive power gating

ParaDox schedules checker cores to favor those with lower IDs (section IV-C), and allows higher ID cores, along with their logs and instruction caches, to be power gated when not in use. We see in figure 12 that while *gobmk*, *sjeng* and *h264ref* make use of all 16 checker cores in times of peak demand, typical usage is much lower; no workload uses more than eight checker cores aggregated across the entire execution. Even though the area overheads of this are still less than a third of the rest of the core [8], even relative to a small main core such as the Cortex A57, this suggests that this could be reduced by half through sharing checker cores between multiple main cores, without affecting performance.



(a) Wake rate for each of the sixteen checker cores, presented as the width of the green line relative to the yellow background.



(b) Average wake rate for each of the sixteen checker cores.

Fig. 12: Proportion of time each of the 16 checker cores is executing, with aggressive checker gating enabled.

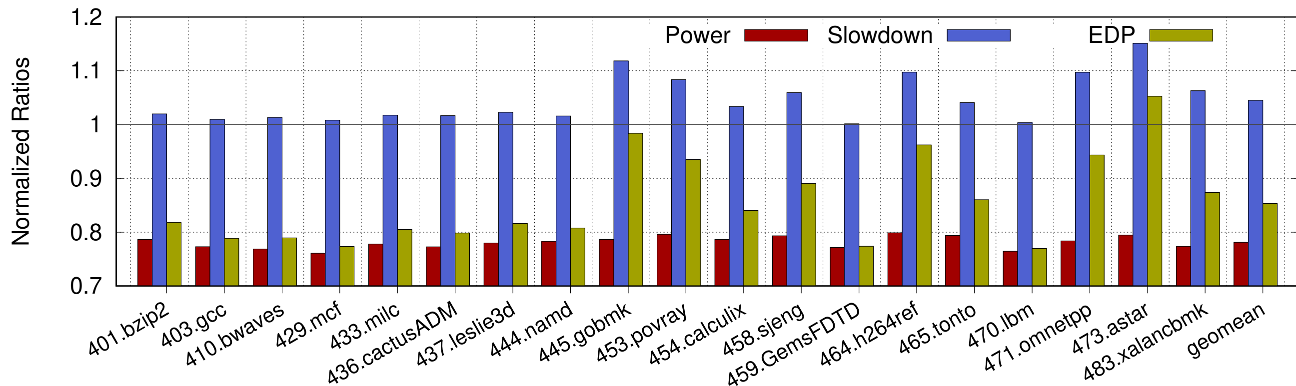


Fig. 13: Power consumption, slowdown and energy delay product on an undervolted system with reliability restored via ParaDox.

E. Power reduction

In figure 13, we take the raw power consumption data for an Arm-based XGene-3 system from Papadimitriou et al. [51], compared with the power consumption of sixteen checker cores, based on public data for small RISC-V rocket cores [4] adjusted [5], [6] to the same 16 nm process as the XGene-3, and with per-workload power gating based on figure 12. The additional power expense of the checker cores is never more than 5% in addition to the underclocked core, and we thus see a 22% reduction in power consumption on average. This 5% figure is from a relatively coarse-grained analysis; alternatives such as McPAT [42] would be more fine-grained, but lack the level of accuracy needed to evaluate the differences between the two very heterogeneous types of core used here. Still, the overall figure is dominated by the savings from undervolting, and so changes in the power consumption of the checker cores would affect the overall saving by only a small factor. With a larger out-of-order main core, this overhead would be reduced further, as superscalar power consumption scales superlinearly with performance, unlike the thread-parallel checker cores.

Still, as ParaDox also introduces performance overhead, we see that the energy-delay product is not always a positive gain. *Astar* suffers from conflict misses in buffered L1 data-cache writes, causing significant slowdown. The technique could be turned off in such a scenario, and voltage margins restored, past thresholds on compute performance on the checker cores. Alternatively, conflict misses in the cache, or the performance impact of ParaDox more generally would need to be improved to have a significant impact on EDP; the power consumption of the checker cores is already minor by comparison. Still, the overall EDP improvement is 15%. By contrast, the EDP of ParaMedic, which does not undervolt, would be 1.08× our baseline, or 1.27× larger than ParaDox.

This figure bears some limitations and does not show the whole potential. First, the values from Papadimitriou et al. [51] are based on error-free undervolting, whereas we see in figure 11 that ParaDox correctly executes at voltages significantly below the point of first error. Second, the analysis assumes a fixed clock frequency for execution, and thus an

element of both slowdown and power reduction. In reality, this is a single point in a complex space; ParaDox could be used by scaling clock frequency as well as voltage. In this scenario, we could instead aim for the original performance level by increasing clock frequency to overcome the slowdown of ParaDox, while still reducing voltage margins to lower than their original level. If we assume power consumption is proportional to $V^2 f$, and f to $V - V_t$ [21], a 4.5% clock frequency increase to mitigate the slowdown could be achieved with around 0.019 V (at a base of .872 V and threshold .45 V [25]), increasing power consumption by 9% relative to the slower case, but reducing it by 15% relative to the voltage-margined baseline. Alternatively, we could target the original power consumption and instead increase performance above the original level, by reducing voltage less and increasing clock frequency further above the safe specified level. Under this scenario, we could increase voltage by 0.06 V from the undervolted 3.2G Hz value, increasing clock frequency by 13% to around 3.6 GHz under the same assumptions. Both this, and the ability to redistribute thermal and power budgets elsewhere in the system, allows ParaDox to have impact on both power consumption and performance, depending on the properties necessary for a given system. Finally, it does not take into account any of the further potential benefits from relaxing voltage spike mitigations [32], [57] by using ParaDox.

F. Summary

ParaDox improves on ParaMedic's ability to deal with high-error scenarios, allowing it to typically achieve the same level of performance at two orders of magnitude higher error rates. By dynamically adjusting its voltage based on observed error rates from applications, ParaDox can deliberately encourage errors to minimize voltage and power consumption, while recovering from any errors and with negligible additional performance impact compared with fault-tolerant error-free execution. We estimate that ParaDox can reduce power consumption by 22%, thus giving energy-delay-product reductions of 15% once its inherent slowdowns are taken into account. In addition, this power consumption can be traded off for restored or improved performance relative to a default-margin baseline, by dynamically scaling frequency and voltage.

VII. Related work

A. Reduced margins

Undervolting can give significant reductions in energy consumption, and conversely, performance improvements. Tovletoglou et al. [67] measure that cutting voltage margins without providing further resilience results in a 20.2% power reduction, whereas Papadimitriou et. al [51], [53] measure savings of 25.2% for an XGene-2 server and 22.3% for an X-Gene 3, and up to 20% energy savings for Intel cores [52]. Leng et al. [41] discover similar levels of savings on GPUs, and Salami et al [60] measure much larger guard bands on FPGAs.

DIVA [11] is an example that, like ParaDox, utilises heterogeneity between checker and main core, with DIVA extracting instruction-level parallelism via a superscalar checker, rather than thread-level parallelism with ParaDox's many-core architecture. EVAL [61] uses a DIVA-like setup to allow aggressive undervolting, like ParaDox. Still, unlike DIVA, ParaDox does not require ECC on architectural state, and thus avoids affecting the critical paths of modern superscalars. Razor [18], [24] is an architecture that augments critical-path flip-flops with shadow latches that are guaranteed against delay errors: this can protect against some forms of delay-based soft error, though with vulnerabilities to metastability [15] and with a high inner-complexity cost. Tan et al. [65] experimentally measure the dependency of soft error rates with undervolting, with a software approach to provide fault tolerance for linear algebra and matrix applications.

Other fault models come from designs that aim at preventing errors altogether. Jiao et al. [36] focus on timing errors, predicting them in order to adapt the timing margins before any actual error happens. Similarly Lefurgy et al. [40] implement an adaptive voltage and frequency scaling mechanism that dynamically measures and adapts timing margins on an IBM POWER7 server, with Zu et al. [70] going further by dynamically colocating workloads based on error characteristics. Bacha and Teodorescu use On-Chip ECC as an early indicator of voltage errors [12], [13]. Chang et al. [23] design mechanisms to trade off supply voltage margins for increased latency on DRAM chips. Krimer et al. [38] split SIMD lanes to reduce the probability of common-mode errors under timing violations. Parasyris et al. [54] investigate near-threshold voltages [37] and the probability of appearance of the first error as undervolting is applied. Chandramoorthy et al. [22] design SRAM mitigations for low-voltage error-tolerant accelerators, rather than the errors in general-purpose compute that are targeted by ParaDox, which require more comprehensive protection. Bertran et al. [16] give a technique for mapping voltage-spike stress marks.

B. Error resilience

Handling faults that only appear at the hardware level is an old problem, starting with the error-prone vacuum-tubes that were used in early computers. Error-resilience mechanisms exist in various forms [27]: some rely on built-in self-test [45], others on dynamic verification of invariants like in Argus [46], but most hinge on some form of redundancy. Static information redundancy, as used for ECC [48], does not protect against combinational-logic faults that can arise at any point during processor runtime. Those require a repetition of the program execution itself, using a combination of space or time redundancy—the former consists in duplicating the components whereas the latter re-executes on the same hardware.

Various design points of the microarchitectural landscape have been explored to provide such fault tolerance. Most solutions can be characterized by their position on the trade-off between coverage, additional hardware complexity and performance and power overhead. Coverage by itself is a complex problem, because of the variety of possible errors that may hit the hardware. Aggarwal et al. [7] propose a design for compartmentalising the faulty components: this idea allows both Romanescu and Sorin [58] and Gupta et al. [29] to offer

architectures in which the valid components that are part of faulty pipelines can be reused to form alternative functioning pipelines. To mitigate complexity, various microarchitecture designs repurpose components that already exist to provide error resilience. StageWeb [28], which refines StageNet [29], illustrates this point by leveraging the redundancy present in chip multiprocessors to recreate a functioning pipeline out of several ones broken at different points, providing tolerance to hard faults and graceful performance degradation. Reliability-Aware Scheduling [50] shows that scheduling to heterogeneous systems based on the type of operation can reduce error rates. Still, since we wish to eliminate errors completely, ParaDox must run on multiple types of heterogeneous core simultaneously.

The line between spatial and temporal redundancy becomes blurred with multicore error resilience. AR-SMT [59] explores the idea of checking the workload of one core, the "leading thread", by another (or possibly the same) core, the "trailing thread", which benefits from the cache warm-up and branch prediction of the leading thread. This design is difficult to implement on hardware however, according to Mukherjee et al. [49], which prompts LaFrieda et al. [39] to design Dynamic Core Coupling, in which the leading and trailing threads are dynamically paired at run-time: this allows the same hardware parts to act as main and checker cores, and it provides the flexibility to perform on-demand TMR in case a hard fault is suspected, by assigning more than one trailing thread to a leader thread. ParaMedic [10] builds on top of its core error-detection scheme [8] which provides the same guarantees at a lower cost via architectural heterogeneity and parallelism. A similar form of slice-based parallelism, used here for fault tolerant executions, has previously been suggested by Zilles and Sohi [69] to verify value-predicted applications.

VIII. Conclusion

ParaDox adapts low-cost fault-tolerance hardware to work in error-seeking environments, by dynamically adjusting voltage below margins to lower energy consumption while preventing the propagation of errors, by re-checking this unreliable computation on multiple parallel checker cores. Despite being based on a delay-tolerant reliability mechanism [10] where errors are assumed to be exceptional, we have shown that we can adapt this delay tolerance and design new mechanisms to efficiently deal with high error frequencies: ParaDox gives typical energy-delay-product reductions of 15%.

This work has further interesting implications for the processor industry. By using hardware fault tolerance to improve performance and energy efficiency, we are likely to see a convergence of high-reliability and commodity hardware. This is likely to significantly lower the cost of high-reliability systems, driving their uptake to new markets. ParaDox has the potential to improve reliability, energy-efficiency and performance across the board.

References

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

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

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

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

[5] https://www.tsmc.com/english/dedicatedFoundry/technology/logic.htm.

[6] https://www.globalfoundries.com/sites/default/files/product-briefs/pb-28slp.pdf.

[7] 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.

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

[9] S. Ainsworth, T. C. Grocutt, and T. M. Jones, "Main processor error detection using checker processors," Mar. 19 2020, US Patent App. 16/338,757.

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

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

[12] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors," in *ISCA*, 2013.

[13] A. Bacha and R. Teodorescu, "Using ECC feedback to guide voltage speculation in low-voltage processors," in *MICRO*, 2014.

[14] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *Digest. International Electron Devices Meeting,*, Dec 2002.

[15] S. Beer, M. Cannizzaro, J. Cortadella, R. Ginosar, and L. Lavagno, "Metastability in better-than-worst-case designs," in *ASYNC*, 2014, pp. 101–102.

[16] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. Carey, R. F. Rizzolo, and T. Strach, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *MICRO*, 2014.

[17] 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, no. 2, 2011.

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

[19] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[20] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *DAC*, 2003, pp. 338–342.

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

[22] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *HPCA*, 2019, pp. 147–158.

[23] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. OConnor, H. Hassan, and O. Mutlu, "Understanding reduced-voltage operation in modern DRAM devices: Experimental characterization, analysis, and mechanisms," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, Jun. 2017.

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

[25] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, Oct 2017.

[26] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, March 2016.

[27] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

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

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

[30] 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.

[31] 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.

[32] Z. Hadjilambrou, S. Das, P. N. Whatmough, D. Bull, and Y. Sazeides, "GeST: An automatic framework for generating CPU stress-tests," in *ISPASS*, 2019.

[33] K. W. Harris, "Asymmetries in soft-error rates in a large cluster system," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 336–342, Sep. 2005.

[34] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[35] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The Arm triple core lock-step (TCLS) processor," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, pp. 7:1–7:30, Jun. 2019.

[36] X. Jiao, A. Rahimi, Y. Jiang, J. Wang, H. Fatemi, J. P. de Gyvez, and R. K. Gupta, "CLIM: A cross-level workload-aware timing error prediction model for functional units," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 771–783, June 2018.

[37] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (NTV) design: Opportunities and challenges," in *DAC*, 2012.

[38] E. Krimer, P. Chiang, and M. Erez, "Lane decoupling for improving the timing-error resiliency of wide-SIMD architectures," in *ISCA*, 2012.

[39] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *DSN*, June 2007, pp. 317–326.

[40] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, J. B. Carter, and R. W. Berry, "Active guardband management in Power7+ to save energy and maintain reliability," *IEEE Micro*, vol. 33, no. 4, 2013.

[41] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach," in *MICRO*, 2015.

[42] 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.

[43] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *ATC*, 2010, pp. 75–88.

[44] 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.

[45] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design Test of Computers*, vol. 2, no. 2, April 1985.

[46] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO*, Dec 2007, pp. 210–222.

[47] 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*, vol. 5, no. 3, pp. 329–335, Sep. 2005.

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

[49] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *ISCA*, 2002.

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

[51] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus," in *HPCA*, 2019.

[52] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, C. Magdalinos, and D. Gizopoulos, "Voltage margins identification on commercial x86-64 multicore microprocessors," in *IOLTS*, 2017, pp. 51–56.

[53] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore CPUs," in *MICRO*, 2017.

[54] K. Parasyris, P. Koutsovasilis, V. Vassiliadis, C. D. Antonopoulos, N. Bellas, and S. Lalis, "A framework for evaluating software on reduced margins hardware," in *DSN*, 2018.

[55] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *DSN*, 2014, pp. 622–629.

[56] E. L. Petersen, P. Shapiro, J. H. Adams, and E. A. Burke, "Calculation of cosmic-ray induced soft upsets and scaling in VLSI devices," *IEEE Transactions on Nuclear Science*, vol. 29, no. 6, pp. 2055–2063, Dec 1982.

[57] M. D. Powell and T. N. Vijaykumar, "Pipeline damping: A microarchitectural technique to reduce inductive noise in supply voltage," in *ISCA*, 2003.

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

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

[60] B. Salami, O. S. Unsal, and A. C. Kestelman, "Comprehensive evaluation of supply voltage underscaling in FPGA on-chip memories," in *MICRO*, 2018.

[61] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "Eval: Utilizing processors with variation-induced timing errors," in *MICRO*, 2008, pp. 423–434.

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

[63] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *DSN*, 2002.

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

[65] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *IPDPS*, 2015.

[66] R. Thomas, K. Barber, N. Sedaghati, L. Zhou, and R. Teodorescu, "Core tunneling: Variation-aware voltage noise mitigation in GPUs," in *HPCA*, 2016.

[67] K. Tovletoglou, L. Mukhanov, G. Karakonstantis, A. Chatzidimitriou, G. Papadimitriou, M. Kaliorakis, D. Gizopoulos, Z. Hadjilambrou, Y. Sazeides, A. Lampropulos, S. Das, and P. Vo, "Measuring and exploiting guardbands of server-grade ARMv8 CPU cores and DRAMs," in *DSN-W*, 2018.

[68] Y. R. Yang and S. S. Lam, "General aimd congestion control," in *ICNP*, Nov 2000.

[69] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *ISCA*, 2001.

[70] Y. Zu, C. R. Lefurgy, J. Leng, M. Halpern, M. S. Floyd, and V. J. Reddi, "Adaptive guardband scheduling to improve system-level efficiency of the POWER7+," in *MICRO*, 2015.