Mechanisms and Evaluation of Cross-Layer Fault-Tolerance for Supercomputing

Chen-Han Ho* Marc de Kruijf* Karthikeyan Sankaralingam*
Barry Rountree† Martin Schulz† Bronis R. de Supinski†
*University of Wisconsin-Madison †Lawrence Livermore National Laboratory
chen-han@cs.wisc.edu dekruijf@cs.wisc.edu karu@cs.wisc.edu rountree4@llnl.gov schulzm@llnl.gov bronis@llnl.gov

***Abstract*—Reliability is emerging as an important constraint for future microprocessors. Cooperative hardware and software approaches for error tolerance can solve this hardware reliability challenge. Cross-layer fault tolerance frameworks expose hardware failures to upper-layers, like the compiler, to help correct faults. Such cooperative approaches require less hardware complexity than masking all faults at the hardware level and are generally more energy efficient.**

This paper provides a detailed design and an implementation study of cross-layer fault tolerance for supercomputing. Since supercomputers necessarily involve large component counts, they have more frequent failures than consumer electronics and small systems. Conventionally, these systems use redundancy and checkpointing to achieve reliable computing. However, redundancy increases acquisition as well as recurring energy costs. This paper describes a simple language-level mechanism coupled with complementary compilation and lightweight hardware error detection that provides efficient reliability and cross-layer fault-tolerance for supercomputers. Our evaluation focuses on strong scaling problems for which we can trade computing power for redundancy. Our results show a range of 1.07x to 2.5x speedup when employing cross-layer error-tolerance compared to conventional full dual modular redundancy (DMR) to contain all errors within hardware. Further, we demonstrate the approach can sustain 7% to 50% lower energy. The most important result of this work is qualitative: we can use a simplified hardware design with relaxed architectural correctness guarantees.

I. Introduction

Manufacturing and process scaling are providing significant challenges in producing reliable transistors for future technologies. Many academic experts, industry consortia and research panels have warned that future generations of silicon technology are likely to be significantly less reliable [56]. A recent exascale study [6] and the Computing Community Consortium Visioning Study [1] conclude that handling reliability will probably become a first-order constraint.

Currently, the prevalent approach to deal with reliability in high performance computing (HPC) is software-based checkpointing and roll-back recovery, as shown in Figure 1(a), which addresses the massive but distributed parallel execution and communication in HPC applications. Entering the exascale era, this conventional system-level fault tolerance in HPC systems faces significant challenges. Massive system scales will lead to sharp drops in system availability even if nodes are highly reliable. For example, when using double-bit ECC memory, which has a mean time to failure of 170 years, a 100,000 memory-module system has a mean time to failure (MTTF) of 20 hours [13]. Managing such frequent failures in software only with system-level or application-level coarse granularity checkpoints incurs high overhead.

DMR-based systems: Borrowing from the literature and techniques on high-availability systems that preserve the software's abstraction of perfect hardware, dual-modular redundancy (DMR) provides a viable solution. For example, the HP NonStop Advanced Architecture uses redundant hardware components to build the system [7]. This solution addresses the availability problem and is widely used in domains like database systems. Figure 1(b) shows an example system that uses dual modular redundancy with hardware checkpoints. Basically, every thread in the system is accompanied by a shadow thread that runs on a different core. These two cores are considered one node, and the shadow core is not exposed to the system. Before the state is committed, the hardware comparator checks if a fault has occurred, and restarts from a hardware checkpoint if needed. An additional hardware datapath is employed for fast comparison of results and checkpointing of stateful hardware structures. This system resolves failures before the software is exposed to them. The DMR based system has nearly perfect coverage of failures, and maintains high availability for each node in the system.

Other novel approaches: To overcome DMR overheads, many have considered hardware fault detection [30], [35], [38], [43], [52] and recovery [3], [11], [41], [53]. Researchers have also explored architectural pruning [36] and timing speculation [15], [20], [21] to mitigate chip design and manufacturing constraints. We cannot directly apply these techniques in the context of HPC systems for three reasons: i) The massive number of nodes; ii) The significant modifications to the processor that these hardware-based techniques require; and iii) The lack of 100% coverage. The modifications are inconsistent with the trend towards hardware simplification to increase energy efficiency [4], [23] while Nomura et al. [40] show these techniques trade-off fault coverage for overhead, or only apply to single stuck-at-faults, and that practical use requires 100% coverage. Thus, we are left with DMR as the only solution *if we want to maintain the perfect hardware abstraction for software*. Wells et al. in their mixed mode multicore [60] and LaFrieda et al. [27] make a similar case and describe example systems. Hence, we use DMR as our baseline of comparison.

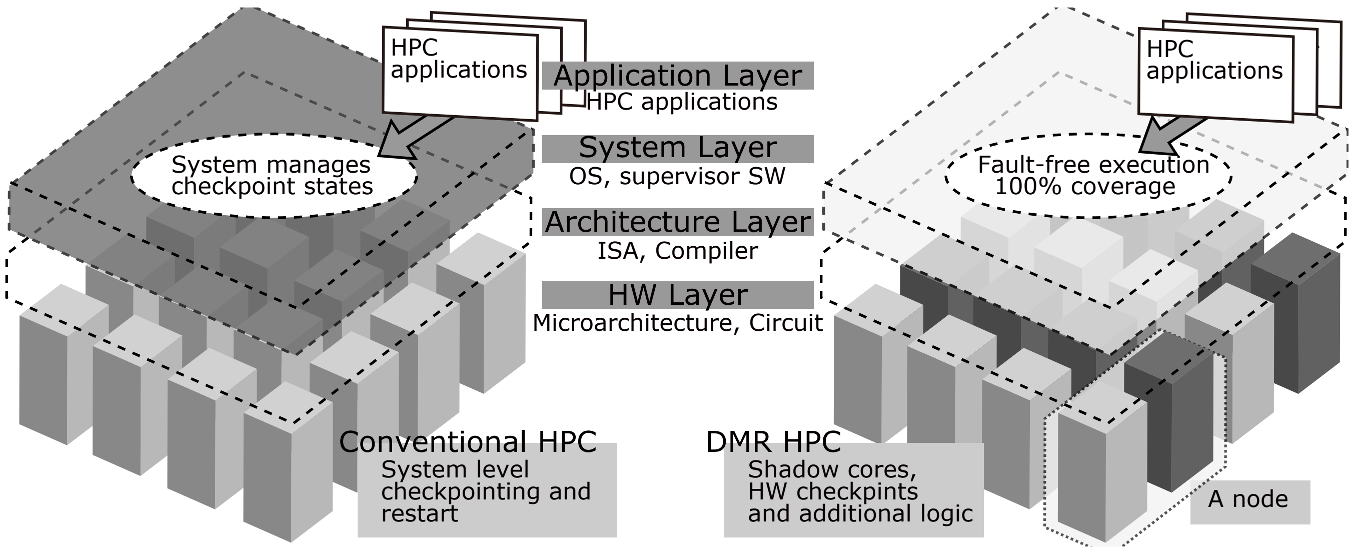
HPC applications
System manages checkpoint states
Application Layer
HPC applications
System Layer
OS, supervisor SW
Architecture Layer
ISA, Compiler
HW Layer
Microarchitecture, Circuit
Conventional HPC
System level checkpointing and restart
HPC applications
Fault-free execution 100% coverage
DMR HPC
Shadow cores, HW checkpints and additional logic
A node

Fig. 1. Fault Tolerance in HPC

Cross-layer: To avoid DMR overheads, we require a cross-layer approach that flexibly provides reliable hardware only when necessary and efficient forms of detection and recovery, *breaking the perfect hardware paradigm*. Examples include: i) language-exposed techniques – EnerJ [48], Flicker [32], Relax [12], M [55], Containment-Domains [57], and Horning's pioneering work [24]; ii) compiler transformations – code-perforation [2], Onward [45], and Green [5]; and iii) architectures – Relax [12], Truffle [16], stochastic processing stochastic, Encore [19], Shoestring [18], and software-based error correction codes [50]. Thus far, these techniques have not been studied in the context of HPC or investigated in the context that the final application is not "naturally" error-tolerant.

Cross-Layer Fault-Tolerance in HPC: In this paper, we evaluate a cross-layer fault tolerance approach for HPC systems to understand the tradeoffs. We use *Relax*, which allows programmers to identify code regions that can tolerate low-reliability explicitly and uses a simple extension to propagate hardware error information to software. It detects failures in hardware, but tolerates faults in software. Thus, it provides the view of perfect hardware, while allowing an implementation that uses inexpensive, low overhead hardware combined with modest programmer effort. Further, recurring energy costs like running redundant copies to handle reliability can also be curtailed. Relax has conceptual similarities to other cross-layer frameworks so our results should hold for them also.

Specifically, *Relax* provides a novel tradeoff in terms of hardware costs and application execution time with a cross-layer approach. We analyze four supercomputing applications across a diverse parameter space in regions that cover between 40% to 90% of the application's execution time. Using a novel fault-injection methodology, we run these applications on real problem sizes on a real HPC system. Quantitatively, we show that the saving of hardware resources by employing Relax provides 1.07x to 2.5x improvement in performance or 7% to 50% energy reduction compared to DMR.

The main contributions of this paper are the following:

- The first end-to-end study of cross-layer fault-tolerance in a supercomputing deployment;
- Identification of key differences between supercomputing and emerging applications for reliability management;
- A demonstration of explicit reliability management to reduce supercomputing hardware correctness requirements;
- An evaluation of how to apply the Relax language-level constructs to HPC benchmarks;
- A quantification of the execution time, energy, and capital cost benefits of Relax in a real supercomputer.

The remainder of this paper is organized as follows. Section II shows the basic insights of exposing hardware faults to applications and background on Relax. Section III describes our Relaxed HPC system architecture and discusses design tradeoffs and implications. Sections IV and V describe our experimental methodology and quantitative results.

II. Relax Background

We briefly describe the background and software interface of the previously proposed Relax framework [12]. As mentioned before other cross-layer frameworks, such as EnerJ, Flicker, and M, have similar semantics.

In the Relax architectural framework, relax "blocks" mark software regions (in source code) that are allowed to experience a hardware fault, and optional recover blocks specify recovery code to execute if a fault occurs. Code Listing 1 shows a C function that returns the *sum of absolute differences* over the array inputs `left` and `right`. We explain two simple software-level recovery use cases based on this code that is adapted from the x264 video encoding application.

Many large-scale scientific applications employ similar reduction computations throughout their execution as in our x264 example. x264 uses a two-dimensional version of this function to search for a predicted frame macroblock's most similar reference frame macroblock. The function measures similarity by performing a pixel by pixel comparison over

Code Listing 1 The *sum of absolute differences* code example that is the basis for all use cases.

```
int sad(int *left, int *right, int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i)
    sum += abs(left[i] - right[i]);
  return sum;
}
```

	Retry
Coarse-grained	`int sad(int *left, int *right, int len) {` `relax (rate) {` `int sum = 0;` `for (int i = 0; i < len; ++i)` `sum += abs(left[i] - right[i]);` `} recover { retry; }` `return sum;` `}` **Use Case 1 (CoRe)**
Fine-grained	`int sad(int *left, int *right, int len) {` `int sum = 0;` `for (int i = 0; i < len; ++i)` `relax (rate) {` `sum += abs(left[i] - right[i]);` `} recover { retry; }` `return sum;` `}` **Use Case 2 (FiRe)**

TABLE I
OUR USE CASES CLASSIFIED BY GRANULARITY AND RECOVERY BEHAVIOR.

two macroblocks. A high similarity presents redundancy that we can exploit to minimize the amount of information encoded. The overall process, *motion estimation*, supports higher data compression rates. We consider two recovery behaviors: coarse-grained (CoRe) and fine-grained (FiRe) retry; we omit the discard behavior [12] since many scientific applications cannot use it. Table I illustrates the resulting taxonomy.

*Use Case 1: Coarse-Grained Retry (*CoRe*)* We can implement coarse-grained retry (CoRe) through relax and recover blocks as in the top half of Table I. We wrap all code except the return statement in a relax block. We protect code *susceptible to failure* by a relax block, where a hardware fault detected inside the block constitutes failure. The variable `rate` specifies a relax block's *probability of failure*. If a failure occurs, control transfers to the recover block, which in this case contains a retry statement that leads to re-execution of the relax block.

The necessary insight for CoRe is that the function in Code Listing 1 has no memory side-effects and thus has no state, beyond its input state, that we must recover if a failure occurs inside the function. If one does, we can simply jump back to the beginning of the function, with the guarantee that the code has not clobbered the input registers. The compiler transparently enforces this guarantee whenever such a control path exists, thereby effectively implementing a software checkpoint. However, the checkpoint is extremely lightweight: the compiler only saves strictly required state.

*Use Case 2: Fine-Grained Retry (*FiRe*)* An alternative to CoRe is to retry at a finer granularity to minimize the amount of wasted work on failure. We can simply move the relax block into the loop to minimize the wasted work as in the bottom half of Table I. We make each iteration a relax block in this case. Since the block's last instruction is the accumulation into `sum`, we can immediately overwrite the old value of `sum` when the block terminates and, thus, require *no* state saving.

III. Relaxing High Performance Computing

First we provide the intuition of why one can employ the Relax framework to improve the hardware-based fault tolerance in an HPC system. We then describe our Relax-HPC system and detail its architectural and programming benefits. This section concludes with a broad discussion of relaxed HPC systems and related design questions.

A. Intuition and Insight

For a strong scaling problem, the hardware-based approach trades execution time for redundant hardware to provide fault tolerance. Acquisition costs in terms of number of cores required and recurring costs in terms of energy are two straightforward metrics. We explore the emerging space of the potential benefits of a cross-layer approach, Relax. In general, the Relax framework can provide similar fault tolerance while trading acquisition and energy costs for application re-execution. The Relax framework relies on the programmer's wisdom to create Relax blocks that protect application regions. In these regions, the hardware does not need to provide any protection. Crossing the application layer, Relax reduces processor design complexity significantly since designers do not have to guarantee architectural correctness in relax blocks.

B. Relax-HPC System Architecture

Figure 2 shows a Relax-HPC system. With applications that use relax blocks, the hardware does not require DMR's architecturally correct execution guarantees. Instead the hardware only must guarantee that faults are detected if they occur and that execution past a relax region does not proceed until the absence of faults is confirmed. Thus, a lightweight technique to detect faults is sufficient. Depending on the fault model, we can use several underlying detection techniques. For example, we can detect hardware timing faults or intermittent faults with canary circuits [10], timing detectors [58], or the Razor flip-flop [15], and transient faults with the BISER latch [37]. The exact detection technique is unimportant; we only assume that it has low overhead. The key point is that recovery comes through software re-execution with no hardware effort. While we use the Relax framework, because of this decoupling, our results apply to other approaches like Truffle/EnerJ [16], [48].

C. Architecture and Benefits

Compared to our DMR baseline, the main area and energy savings come from requiring fewer execution resources. If we consider strong scaling of HPC applications then we use the additional resources to solve the problem faster while, if

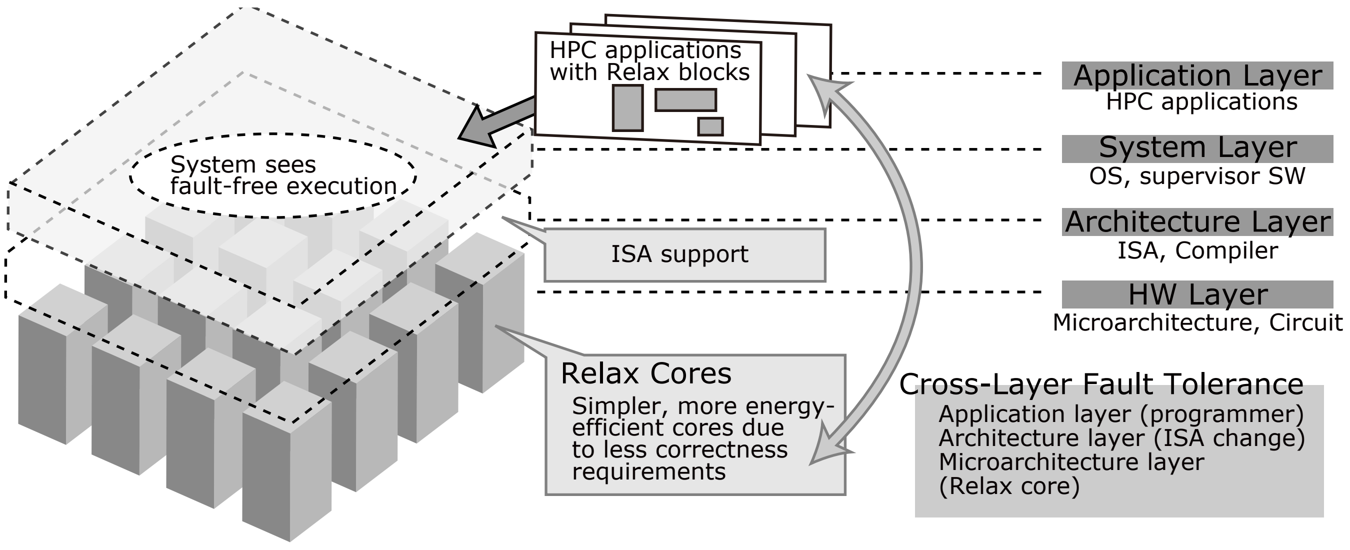
HPC applications with Relax blocks
System sees fault-free execution
ISA support
Relax Cores
Simpler, more energy-efficient cores due to less correctness requirements
Application Layer
HPC applications
System Layer
OS, supervisor SW
Architecture Layer
ISA, Compiler
HW Layer
Microarchitecture, Circuit
Cross-Layer Fault Tolerance
Application layer (programmer)
Architecture layer (ISA change)
Microarchitecture layer
(Relax core)

Fig. 2. Relax in HPC

we consider weak scaling, then we can solve larger problem sizes. In this work's evaluation, we focus on strong scaling and examine the reduced execution time from more available cores, and the improvement in energy that comes from Relax+software recovery compared to DMR execution.

D. Programming for a Relax-HPC System

Relax's language-level constructs are intuitive and its compilation framework generates code that can automatically recover from a fault. Other frameworks like Flicker [32] and EnerJ [48] also are designed to be intuitive and easy to use.

We extend the Relax framework to handle interprocess communication, which is implemented through MPI for all applications that we consider. To simplify this study, we carefully construct all relax regions so that no MPI operations are active during them. This choice indirectly models the error propagation, but effectively assumes faults never occur within MPI calls. We essentially assume that some other mechanism, such as FT-MPI [17], ensures fault tolerant MPI activity.

We leave automating the process of determining relax regions as future work. The key challenge is to transfer program control at the end of a region to some valid point from which re-execution will produce the correct result if a fault occurs. For now, we determine this point in a two-step process. First, we use conventional performance profiling to determine hot code regions. Second, we convert a subset of those regions into relax blocks. Typically, this process is straightforward, and we can easily relax applications with which we are unfamiliar.

E. Applications

Prior work has explicitly applied Relax to naturally error-tolerant applications. While HPC applications have similarities, they also exhibit important differences.

Similarities to naturally error tolerant applications: Error tolerance is a useful property. However, non-error tolerant instructions are typically interspersed with error-tolerant computation instructions, which complicates exploiting them. Relax leverages the ability to recover by discarding or re-executing groups of instructions at a coarse granularity. We find that this property holds for HPC applications, which are compute-dominated, similarly to emerging applications.

Differences: We identify fundamental differences with respect to the Relax language construct between HPC applications and naturally error tolerant ones. First, the scale of parallelism in these applications is large. In the PARSEC suite, which is a representative benchmark suite for emerging applications, typically 15% to 25% of the application is serial [8], [9]. HPC applications have much smaller serial phases. Second, communication between tasks introduces error propagation between cores and processes, which complicates the fault recovery approach of reexecution within a process.

*Showing that HPC applications do **not** require strict instruction-granularity correctness is one of the important contributions of this work.*

F. Discussion

In this section, we discuss some natural questions that arise for this unconventional system.

How to handle non-relaxed code regions? As described above and quantified in Section V, our relax regions do not cover some (small) application code regions. For these parts, the hardware must guarantee architecturally correct execution. We assume an execution model that is similar to the mixed mode multicore [60] and DCC [27] approaches: the hardware explicitly enters and leaves DMR mode. Alternatively, we can use any of the following software-based approaches. Software instruction-level DMR for these "critical" regions as developed by Reis et al. can detect errors at load, store, and control flow boundaries [42]. Depending on the need for recovery, instruction-level TMR can add a third copy of each non-memory instruction and uses majority voting before load and store instructions to detect and to correct failures [11]. If applied in the context of the Truffle/EnerJ [16], [48], the non-relaxed code would always execute in the precise pipeline. We leave the application of relax regions within an MPI implementation for future work.

What about spontaneous bit-flips in memory or processor control signals? Any cross-layer technique must make assumptions about the faults that can occur. In our Relax-HPC system, we assume the core has ECC for storage and "hardened" logic for control-flow instructions thus making program control-flow always correct. Thus we assume the Locally Corruptible Error model [54], as do Encore [19], Shoestring [18], stochastic processing [49], and architectural and language approaches for approximate/precise computing [16], [48].

If you have fault-detection, why not immediately retry? Our Relax-HPC system intentionally decouples fault detection from fault recovery to model various possible recovery scenarios. This model has been assumed in various systems including Relax [12], Encore [19], and ShoeString [18]. Further, "immediate retry" in the processor may require the addition of checkpoints, buffering structures and staging for temporary results during detection. By decoupling, we assume a more relaxed model that allows incorrect results to be written. With algorithmic detection techniques [51], we have no natural instruction-precise "immediate" retry so we must identify a well defined point. Also, this decoupling allows our results to consider techniques like Truffle [16], which steers instructions into low-energy approximate or high-energy precise pipelines. If applied to this model, all instructions in a relax block would execute in the approximate pipeline. Finally, by creating large blocks of code annotated with a fault-rate that they can tolerate, we avoid re-execution for many instructions, whereas immediate retry would re-execute each.

What if fault-detection is not 100%? Improved fault detection techniques are an active area of research that is orthogonal to our work. For this work we assume fault-detection is 100%, which is reasonable in many situations. For permanent faults and single stuck-at faults, techniques like Sampling-DMR provide this guarantee. For transient faults, 100% detection is possible for storage under certain fault assumptions (like single-bit flip) by using ECC. For logic faults, fault detection merely reduces FIT rates (Argus [34], or BISER [37] latch).

Why compare to DMR? As outlined in the introduction, today's HPC systems do not use DMR. However, given fault-rates and projections for the scale of future HPC systems, current approaches cannot sustain the required MTTF. Among the approaches that **do not** expose hardware faults/errors to applications, DMR is the only candidate that can handle future project fault rates. Hence we pick DMR as the baseline, similarly to evaluations of other researchers [27], [60].

How does this approach relate to checkpoint/restart? Application-level checkpoint/restart will remain necessary in future HPC systems to handle catastrophic failures that impact multiple nodes. The frequency of checkpointing depends on the failure probabilities. Without techniques to reduce the projected failure probabilities of individual nodes, checkpointing frequency will imply prohibitive costs. Our Relax-HPC system effectively reduces the failure probability of individual nodes, thus reducing the required checkpoint frequency.

Novelty compared to prior work? No previous cross-layer tolerance study has examined HPC workloads that require strict correctness guarantees. Further, all prior work have been simulation studies that do not employ real supercomputers. Our demonstration that HPC workloads tolerate retry and discard behavior is an important result. Further, our evaluation shows how cross-layer fault tolerance can produce better energy efficiency and support larger problem sizes, thus quantifying its execution time, energy, and capital cost benefits on *a real supercomputer*. Our work analyzes three real workloads on a real supercomputer deployed at LLNL.

IV. Methodology

A. Overview

To evaluate the cross-layer fault tolerance approach, we compare Relax recovery using lightweight fault detection with checkpoint recovery using dual modular redundancy at the core level. This baseline configuration represents a typical approach for fast fault recovery with moderate architecture modification. In contrast, the Relax hardware is simplified, but requires programmers to define the application behavior for when a fault happens.

In this study, we use Relax's retry recovery behavior to recover the faulty execution. This re-execution results in longer execution times. However, compared to the baseline processor, more cores are available for use by the application. If the speedup from the extra cores exceeds the slow down from re-execution, overall performance improves.

In this paper, we use a novel fault injection approach to study a future faulty supercomputer system using today's "perfect" hardware. We use a compiler-based instruction-level fault injection tool to create binaries that emulate faulty behavior, yet can be executed on actual hardware. We extend this LLVM based infrastructure [12] for our supercomputing cluster. We call instrumented binaries *relaxed binaries* to be compared with original unmodified binaries. Both are x86 binaries, with the only difference being that one is instrumented to emulate fault behavior. We use a simple and conservative model that weights execution time to reflect that only part of the application is relaxed even in the relaxed binary because relax regions do not cover 100% of the application. For this study, we use the LLNL Hyperion cluster as our supercomputing environment and perform studies on 128 to 1024 cores.

Fig. 3 shows the complete experimental flow. In the remainder of this section, we show how we choose our experimental benchmarks and input sets. We then discuss the LLVM-based fault injection tool and our evaluation platform and metrics.

B. Benchmarks and Experiments

We focus on application with strong scaling workloads instead of MPI performance testing benchmarks such as Intel's MPI benchmarks, since we are concerned with overall

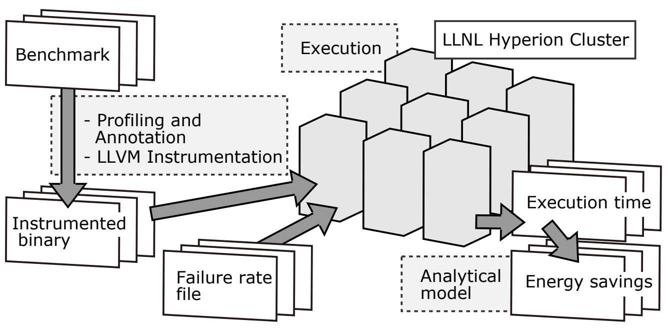

Fig. 3. Evaluation flow

system performance. The LLVM instrumentation tool, which only supports C/C++ benchmarks also limits our benchmark selections. To show representative results, the benchmarks should scale to 1,000 cores or more. In all, we choose AMG (in which we use two different solvers) and IRS from the Sequoia benchmark suite [28], and IS from the NPB benchmark suite [39]. We fix problem sizes regardless of configuration and scale the number of MPI processes to show the performance improvement with scaling in reasonable simulation time. For each benchmark, we experiment using two different configurations, which are as follows:

- Sequoia AMG: PCG and GMRES Solvers (preconditioner conjugate gradient solver and generalized minimal residual solver): $250 \times 250 \times 250$ grid points and $500 \times 500 \times 500$ grid points.
- Sequoia IRS: $5 \times 5 \times 5$ domains and $10 \times 10 \times 10$ domains.
- NPB IS : problem set C and problem set D.

To characterize our results, we compare to uninstrumented fault-free executions to determine overhead and explore speedup relative to the baseline processor. We also use mpiP profiling [59] to examine the trend of time spent in MPI when scaling the number of cores to provide additional insight into scalability of the different options. These profiles reflect the trend of the communication to computation ratio when scaling.

C. LLVM Compiler Instrumentation

To inject faults into our benchmarks, we use a modified version of an existing LLVM instrumentation tool [12]. At compile time, it instruments every LLVM instruction to inject faults at runtime. During runtime, the instrumented code reads a pre-defined failure rate file that specifies the probability of hardware failures across all microprocessor chips in the cluster.

Whenever a fault occurs, execution jumps to the programmer-defined recovery region, which initiates the retry behavior. We use fine-grained retry behavior for all benchmarks, where the largest Relax block contains less then 400 LLVM instructions (348 instructions in AMG). To find the proper region for fault injection and recovery using Relax, we use LLVM profiling to measure the most frequently executed code regions. These code regions represent the computing phase in the supercomputing applications. We measure the overheads of instrumentation by comparing the execution time of instrumented binary (i.e., the relaxed binary) to the reference binary. Across the different benchmarks our LLVM tool adds runtime overheads in the range of 3.4× to 6.2×. This instrumentation overhead does not affect our execution time measurements with fault-free runs or the accuracy of our speedup results. The issue of instrumentation overhead affecting our measurements only arises when communication and computation overlap. Our work does not instrument the MPI calls (because we did not recompile the MPI library). Thus the overhead does not affect our measurements.

Benchmark	AMG_PCG	AMG_GMRES	IRS	IS
Coverage	55.14%	42.47%	52.84%	81.43%
Frequent region	matrix multiply	matrix multiply	matrix multiply	key sort
# of lines changed	$\approx$ 10 in 1 region	$\approx$ 10 in 1 region	$\approx$ 5 in 1 region	$\approx$ 10 in 2 regions

TABLE II
PERCENTAGE OF DYNAMIC INSTRUCTION COVERAGE

D. Evaluation Platform and Metrics

In this study, we ran experiments on the LLNL Hyperion cluster, on which each node has a 2-socket Harpertown quad-core Xeon 2.66GHz with 8GB DRAM and nodes are connected by QDR Infiniband. We map one MPI process to one physical core. The cluster uses the SLURM resource manager, and we use Open MPI 1.4 with IB support to run the benchmarks. We use execution time to measure system performance, since it is the most crucial metric when considering solving real problems on supercomputers. To measure execution time of a Relax HPC cluster, we run the relaxed binary and measure its execution time. To measure execution time of an application on our baseline system, DMR HPC cluster, we run the relaxed binary on our Hyperion cluster but assume no fault injection (since DMR will mask all faults). Compared to the Relax HPC cluster, we provide half the number of nodes, to account for DMR execution. In our results we compare execution time between Relax and the baseline and application speedup when the number of cores changes.

V. Evaluation

In this section, we first discuss how we determine where to place the Relax block regions in supercomputing applications. Next, we show the overhead of Relax recovery in terms of execution time compared to fault-free execution and the improvement compared to DMR execution when employing Relax. Finally, we show the impact of scaling on the execution time under different fault rates, and discuss the energy implications of Relax in supercomputing.

A. Relax Region

The Relax framework supports programmer-defined fault recovery behavior. We use LLVM profiling to capture the most frequently executed code regions, which guides selection of relax regions. Table II shows the results for our benchmarks. Rows 1-3 show the benchmark name, the percentage of dynamic instructions covered by the Relax code regions, and the name of the most frequently executed code region.

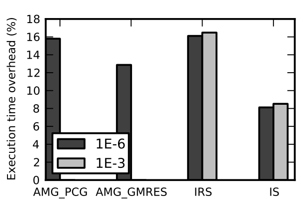

Fig. 4. Execution time overhead (%) compared to fault-free processor

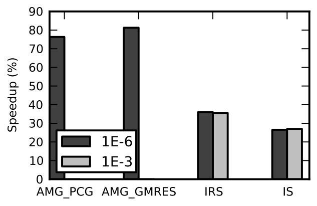

Fig. 5. Speedup (%) compared to the baseline processor

As expected, we observe that these regions constitute the computation phase of the applications. The cost of re-execution is quite small due to the fine granularity. Fine-grain retry provides substantial coverage: among all benchmarks, we cover more than 40% of dynamic instructions. The coverage is dependent on the regularity of the computation pattern in the application. For example, the NAS IS benchmark has the simplest kernel and thus the most coverage, with mainly only the instructions of MPI calls not covered in the instrumentation. Using Relax, cross-layer fault-tolerance can be achieved at a coarser granularity than fine-grained regions. However, when covering large code regions the re-execution cost may be prohibitive in terms of execution time.

B. Overhead of Re-execution

Recovery using re-execution with Relax trades execution time for hardware efficiency. We compare the fault-free execution of the relax binary to its execution time with varying fault rates, all running on the same number of nodes to evaluate the overhead of re-execution. Figure 4 shows this execution time comparison for a 256 node system in terms of different system failure rates for the of entire HPC cluster. The execution time overhead varies from 8% to 16% for a failure rate of 10^{-6}. IRS and IS can tolerate the extremely high failure rate of 10^{-3}. In contrast, with such a high failure rate AMG cannot finish within one week, so we omit those runs from the graph. This behavior is related to the size of the implemented relax region. As mentioned in section IV, AMG has the largest number of instructions in one region. In terms of dynamic instruction count, these regions account for thousands of instructions.

The cost of recovery increases with high failure rates. At 10^{-3} or 10^{-6} failure rate, the cost of coarse-grain recovery methods such as system-level checkpointing is prohibitive because of the total data volume of checkpoints. Cross-layer fault tolerance would be beneficial when the hardware efficiency gains can overcome the overhead of re-execution. In the next section, we discuss the benefit of the Relax framework by comparing it with the baseline system, which uses dual modular redundancy to achieve fault-free execution.

C. Benefit of Hardware Efficiency

The baseline processor uses checkpointing with DMR to provide the illusion of fault-free execution. Removing this redundancy provides additional computation resources, which can be used by the application. Figure 5 shows the improvement of execution time when considering a 256 node Relax HPC system to the baseline DMR HPC system, which effectively has 128 compute nodes in this case. The bars show the speedup measured as a percentage.

In general, the Relax system becomes faster by providing more nodes for strong scaling. This gain occurs despite the additional overhead of re-execution. We observe 27% to 81% improvement in execution time for system fault rates of 10^{-6}. Intuitively, the improvement increases as the fault rate decreases. Thus, we treat the result at a high fault rate of 10^{-6} as a lower bound benefit of the Relax framework.

D. Scalability Study

Figures 6 and 7 show the number of cores (x-axis) versus the execution time (y-axis) for each benchmark at two different problem sizes. Execution time at different failure rates is shown using different markers. The gap between the baseline and Relax execution time represents the potential speed up at a given number of cores. We show an estimation of lower-bound execution time as RelaxL in the figure, which represents the execution time reduction that could be achieved if the entire application was converted into relax regions.

For small problem sizes (Figure 6), the execution time improves significantly with increasing core count. For large problem sizes (Figure 7), the difference of execution time is smaller at a given scale (256 to 1,024 nodes). The effect of a changing failure rate also decreases. As with the small problem sizes, the benchmarks tend to execute faster when using more cores. However, the IRS and IS benchmarks have irregular behavior at 729 and 1,024 nodes, respectively. For IRS, the algorithm is not optimized for strong scaling experiments, and it uses MPI process ranks to map problem domains to cores. For IS, the overhead of MPI communication becomes too high when using 1,024 cores.

The RelaxL configuration shows that the lower-bound execution time of Relax, which can be achieved with more programmer effort, is about a factor of 2 less than the baseline DMR system. Figure 8 shows this best possible speedup across different node sizes. We compare the RelaxL system with various fault-rates to our DMR baseline (assuming no faults) to get an estimate for best case speed up. The benefit from adopting the Relax framework increases as the size of the system increases, except for two anomalies in IRS and IS. The speedup curve is superlinear because the working set is too large at small system scales. In addition, the worst speedup happens when failure rate is high and the system consists of a small number of cores.

Finally, we quantitatively characterize the strong scaling properties of the benchmarks. Figure 9 shows the MPI portion of the execution time when increasing the core count. A strong scaling problem should grow slowly in MPI percentage and keep the communication to computation ratio small. For example, the percent in MPI of AMG only increases by about 1% of total execution time when doubling the cores (increases from 9% to 12% when scaled from 128 to 1,024 nodes). In contrast, IS uses a predetermined problem size and all to all data redistribution. Thus the communication overhead is large (48% to 88%). IRS is not optimized for strong scaling, and

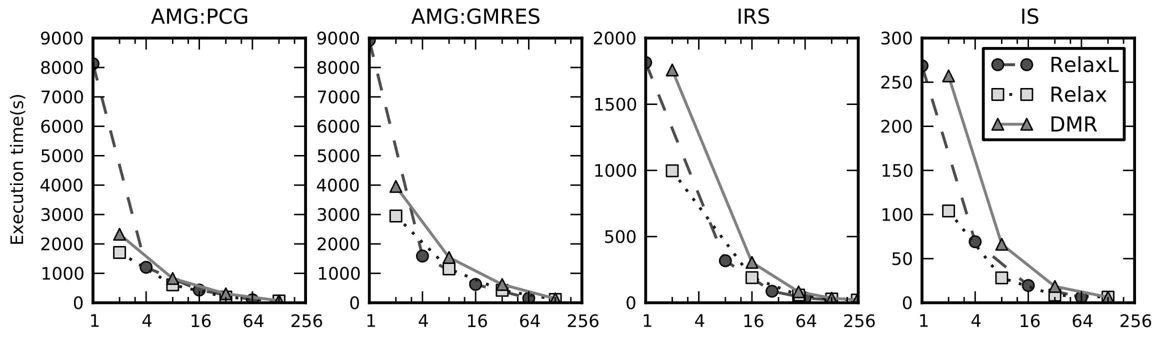

Fig. 6. Execution time (s) versus the number of cores at small problem sizes

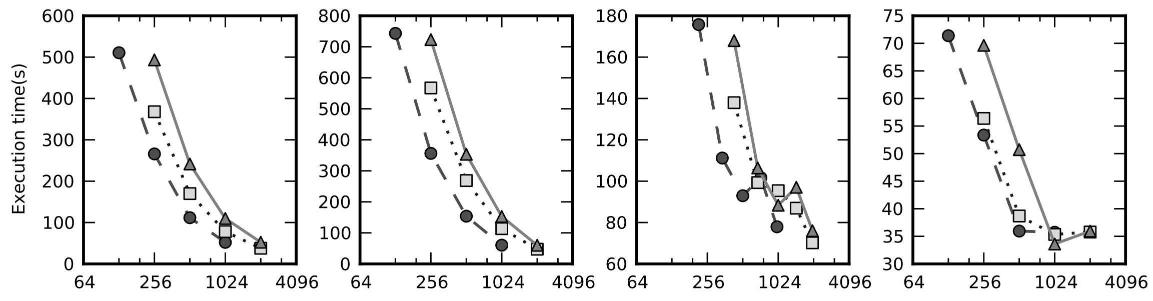

Fig. 7. Execution time (s) versus the number of cores at large problem sizes

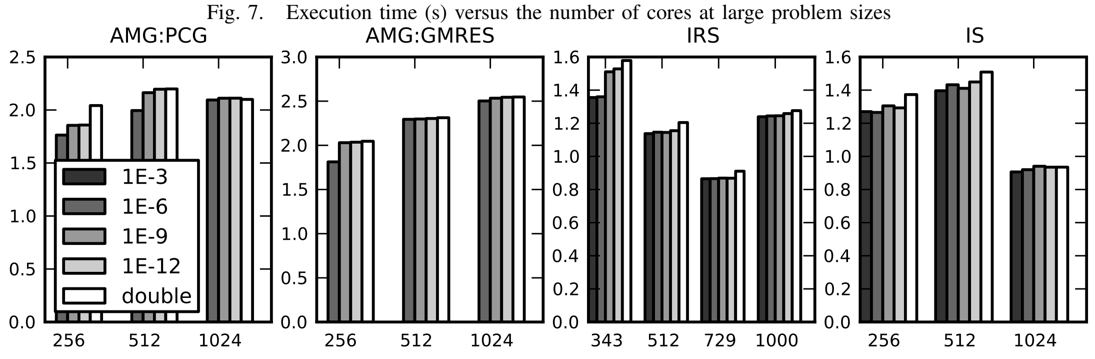

Fig. 8. Speedup when scaling the number of cores (RelaxL compared to DMR baseline).

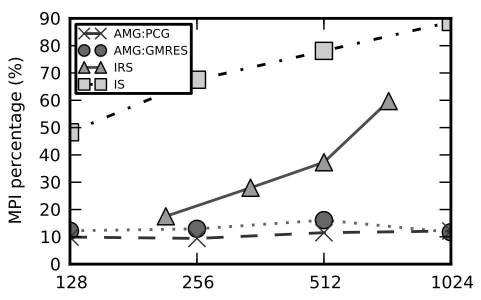

Fig. 9. MPI time percentage (%) at different number of cores

the percent in MPI increases by about 10% of total execution time when scaling the number of cores (17% to 62%).

E. Implications on Energy

To evaluate the power and energy benefit of the cross-layer approach, we use a simple analytical model to calculate the energy improvement. First, the power consumption of a supercomputer can be divided into three parts: the power of the microprocessor chip $P_{processor}$, the power of off chip memory P_{memory}, and the power of I/O, routers and other components in the system P_{system}. The total power of the Relax system can be represented as:

$$P_{Relax} = P_{processor} + P_{memory} + P_{system}$$

A typical power breakdown is 0.56, 0.33, and 0.11 for each part of the system [6]. With dual modular redundancy, the processor uses doubled cores, and each pair of cores shares part of the on chip resources (e.g., caches). We model the power of the DMR system as:

$$P_{DMR} = P_{processor} * (1 + f_{DMR}) + P_{memory} + P_{system}$$

where f_{DMR} is the fraction of power consumed by a redundant DMR core compared to a full-function core. In Figure 10, we show the energy improvement of the Relax framework at different DMR power consumption fractions using:

$$E_{reduction} = 1 - \frac{P_{Relax} \times Relax\ execution\ time}{P_{DMR} \times DMR\ execution\ time}$$

Here, the execution time is the geometric mean of the benchmarks under a failure rate of 10^{-6}. We use the McPAT [31]

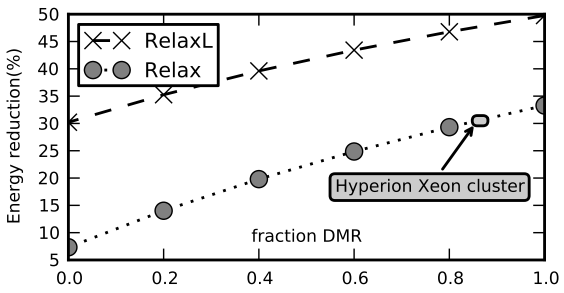

Fig. 10. Energy improvement of Relax

modeling tool to estimate f_{DMR} for DMR pairs that share L1 and L2 caches. As labeled, f_{DMR} is 0.86 for Intel Xeon clusters, and the energy improvement is around 25%. Overall, the improvement in energy using the Relax framework is up to 30%. When considering the best case energy improvement for a completely relaxed application (RelaxL), we see energy improvement varies from 30% to about 50%.

VI. Related Work

In this section, we discuss the related work in fault tolerance for commodity hardware and supercomputing systems. Related cross-layers work was discussed in Section I.

Hardware/software layer: The SWAT system uses hardware and software cooperation to detect hardware failures and to recover faults using hardware checkpoints [30], [47]. It uses symptom detection and checking for invariants to detect faults. To guarantee 100% detection, Relax uses hardware fault detection. If the invariants provide high quality fault coverage, we could use this hardware and software cooperation, which complements our approach, to lower processor complexity even further. Relax recovery techniques require no hardware cost and provide a scalable recovery solution, as our results demonstrate. The scalability of pure hardware checkpointing in supercomputers is an open question.

The Merrimac Streaming Supercomputer [14] uses hardware-software co-design to provide fault tolerance. The authors evaluate software-based fault detection (full-program re-execution, instruction replication, and kernel re-execution) for a supercomputer stream unit. Like SWAT, Merrimac uses hardware checkpointing to restart the computation. In contrast to these mandatory checkpointing frameworks, Relax allows the programmer to define the state required for re-execution. Thus, we can limit saved state to exactly that necessary for re-execution, which the compiler saves automatically, often through register allocation strategies.

Rinard addresses the problem of checkpointing overhead by providing discard behavior at the task level [44]. However this capability requires an explicit definition of the level of correctness required from the program. Its applicability to supercomputing requires further investigation. The Jade language addresses this problem by partitioning parallel applications into tasks and discarding full tasks when a hardware failure occurs [46]. The detection and recovery of this framework happens in language/compiler/software layers. Relax is less expansive in scope and as a result is just a simple language extension so it is easier to adopt. Further, it provides freedom to the program to select relax regions of any granularity. We acknowledge that Relax requires programmer effort since it is an application-based fault-recovery technique.

System layer: Self-aware HPC clusters migrate the execution between nodes to provide adaptive self-healing from failures [29]. Fault detection and recovery happens purely at the system level with no application-level interaction and it has been employed on a commodity Linux cluster.

The Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) [22] provides a framework to handle fault handling and notification. It supports uniform fault handling regardless of the fault source (hardware, operating system, middleware, libraries or application). We currently assume relax constructs only apply to hardware faults. While additional work remains to determine how to react to faults arising in some sources, we could implement the Relax language construct to obtain fault notifications from a system layer fault-tolerance scheme such as CIFTS to tolerate a wider range of faults.

Other: Algorithm-based fault tolerance (ABFT) in general applies application-specific mechanisms for error detection and correction [25]. Kienzle presents an overview of software fault tolerance approaches [26]. Recently, Maruyama et al. presented a mechanisms for software fault tolerance to perform reliable computing with GPUs [33].

VII. Conclusion

Reliability is an important constraint for future systems and consensus is emerging around cross-layer error tolerance for supercomputing applications. This paper provides a detailed quantitative study of one such cross-layer approach: explicit code regions that can be relaxed from architectural correctness. The hardware must only guarantee detection, and recovery is achieved through simple re-execution. We showed in this paper that this simple approach is easy for programmers to use and highly effective. Up to 2x performance improvement was possible for strong-scaling applications with 50% energy reduction. The main implication of this work is that explicit reliability management of applications in supercomputing is tractable and provides tangible benefits. Pushing the reliability limits further through techniques such as overclocking, reducing voltage, and near-threshold operation could provide further benefits by trading reliability for energy efficiency.

Acknowledgments

We thank the anonymous reviewers for their comments. This work has been authored by Lawrence Livermore National Security, LLC under contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting this work for dissemination, acknowledges that the United States Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the disseminated form of this work or allow others to do so, for United States Government purposes. The LLNL release number for this paper is LLNL-CONF-476515. Support for this research was provided by NSF grants CCF-0845751 and CCF-0917238.

References

[1] CCC Visioning Study on Cross-Layer Reliability, http://www.relxlayer.org/.

[2] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical report, MIT, 2009.

[3] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *MICRO-36*, pages 423–434, 2003.

[4] O. Azizi, A. Mahesri, B. C. Lee, S. Patel, and M. Horowitz. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In *ISCA '10*.

[5] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation. In *PLDI*, 2010.

[6] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead, 2008.

[7] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *DSN '05*.

[8] M. Bhadauria, V. Weaver, and S. McKee. Understanding PARSEC Performance on Contemporary CMPs. In *IISWC '09*, pages 98–107.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT '08*.

[10] Ridgetop group, Sentinel Silicon Cells, http://www.ridgetop-group.com.

[11] J. Chang, G. A. Reis, and D. I. August. Automatic Instruction-Level Software-Only Recovery. In *DSN-36*, pages 83–92, 2006.

[12] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *ISCA 2010*.

[13] C. Engelmann, H. Ong, and S. Scott. The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *PCDN '09*.

[14] M. Erez, N. Jayasena, T. J. Knight, and W. J. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. In *SC '05*.

[15] 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 '03*.

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS 2012*.

[17] G. E. Fagg and J. J. Dongarra. Building and Using a Fault Tolerant MPI Implementation. *International Journal of High Performance Computing Applications*, 18:2004, 2004.

[18] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic Soft-error Reliability on the Cheap. In *ASPLOS 2010*.

[19] S. Feng, S. Gupta, A. Ansari, S. Mahlke, and D. August. Encore: Low-Cost, Fine-Grained Transient Fault Recovery. In *MICRO 2011*.

[20] B. Greskamp and J. Torrellas. Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking. In *PACT '07*.

[21] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing Processors for Timing Speculation from the Ground Up. In *HPCA '09*.

[22] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra. CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems. In *ICPP '09*.

[23] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *ISCA '10*.

[24] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A Program Structure for Error Detection and Recovery. In *Operating Systems, Proceedings of an International Symposium*, 1974.

[25] K.-H. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on*, C-33(6):518 – 528, 1984.

[26] J. Kienzle. Software Fault Tolerance: An Overview. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies*, volume 2655 of *Lecture Notes in Computer Science*, pages 641–641. 2003.

[27] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In *DSN '07*.

[28] Lawrence Livermore National Laboratory. The ASC Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks.

[29] C. Leangsuksun, T. Liu, Y. Liu, S. Scott, R. Libby, and I. Haddad. Highly Reliable Linux HPC Clusters: Self-Awareness Approach. In J. Cao, L. Yang, M. Guo, and F. Lau, editors, *Parallel and Distributed Processing and Applications*, volume 3358 of *Lecture Notes in Computer Science*, pages 217–222. 2005.

[30] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *ASPLOS '08*.

[31] 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 '09*.

[32] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flicker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning. Technical Report MSR-TR-2009-138, Microsoft Research, 2009.

[33] N. Maruyama, A. Nukada, and S. Matsuoka. A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs. In *IPDPS '10*.

[34] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *MICRO '07*.

[35] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. *Micro, IEEE*, 28(1):52–59, 2008.

[36] F. Mesa-Martinez and J. Renau. Effective Optimistic-Checker Tandem Core Design through Architectural Pruning. In *MICRO '07*.

[37] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust System Design with Built-In Soft-Error Resilience. *Computer*, 38(2):43–52, 2005.

[38] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multi-Threading Alternatives. In *ISCA '02*.

[39] National Aeronautics and Space Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[40] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + DMR: Practical and Low-overhead Permanent Fault Detection. In *ISCA '11*.

[41] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *ISCA-29*, pages 111–122, 2002.

[42] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software Implemented Fault Tolerance. In *CGO '05*.

[43] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, 2005.

[44] M. Rinard. Probabilistic Accuracy Bounds for Perforated Programs: A New Foundation for Program Analysis and Transformation. PEPM '11.

[45] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and Statistical Analysis for Understanding Reduced Resource Computing. In *Onward!*, 2010.

[46] M. C. Rinard. *The Design, Implementation and Evaluation of Jade: A Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, USA, 1994. UMI Order No. GAX-08439.

[47] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *DSN '08*.

[48] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *PLDI '11*.

[49] J. Sartori, J. Sloan, and R. Kumar. Stochastic Computing: Embracing Errors in Architecture and Design of Processors and Applications. In *CASES 2011*.

[50] P. Shirvani, N. Saxena, and E. McCluskey. Software-Implemented EDAC Protection Against SEUs. *Reliability, IEEE Transactions on*, 49(3):273 –284, Sept. 2000.

[51] J. Sloan, R. Kumar, G. Bronevetsky, and T. Kolev. Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra. In *DSN 2012*.

[52] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *MICRO '06*.

[53] 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-29*, pages 123–134, 2002.

[54] V. Sridharan, D. A. Liberty, and D. R. Kaeli. A Taxonomy to Enable Error Recovery and Correction in Software. In *Workshop on Quality-Aware Design*, 2008.

[55] P. Stanley-Marbell and D. Marculescu. A Programming Model and Language Implementation for Concurrent Failure Prone Hardware. In *Workshop on Programming Models for Ubiquitous Parallelism*, 2006.

[56] A. W. Strong, E. Y. Wu, R.-P. Vollertsen, J. Sune, G. L. Rosa, T. D. Sullivan, S. E. Rauch, and III. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley-IEEE Press.

[57] M. Sullivan, D. H. Yoon, and M. Erez. Containment Domains: A Full-System Approach to Computational Resiliency. Technical Report "TR-LPH-2011-001", UT-Austin, 2011.

[58] J. Tschanz, K. Bowman, S. Walstra, M. Agostinelli, T. Karnik, and V. De. Tunable Replica Circuits and Adaptive Voltage-Frequency Techniques for Dynamic Voltage, Temperature, and Aging Variation Tolerance. In *VLSI Circuits, 2009 Symposium on*, pages 112 –113, 2009.

[59] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/, Apr. 2005.

[60] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-Mode Multicore Reliability. In *ASPLOS -XIV*, 2009.