Globally Precise-restartable Execution of Parallel Programs

Gagan Gupta   Srinath Sridharan   Gurindar S. Sohi

University of Wisconsin-Madison

gagang@cs.wisc.edu   sridhara@cs.wisc.edu   sohi@cs.wisc.edu

Abstract

Emerging trends in computer design and use are likely to make *exceptions*, once rare, the norm, especially as the system size grows. Due to exceptions, arising from hardware faults, approximate computing, dynamic resource management, etc., successful and error-free execution of programs may no longer be assured. Yet, designers will want to tolerate the exceptions so that the programs execute completely, efficiently and without external intervention.

Modern computers easily handle exceptions in sequential programs, using precise interrupts. But they are ill-equipped to handle exceptions in parallel programs, which are growing in prevalence. In this work we introduce the notion of *globally precise-restartable* execution of parallel programs, analogous to precise-interruptible execution of sequential programs. We present a software runtime recovery system based on the approach to handle exceptions in suitably-written parallel programs. Qualitative and quantitative analyses show that the proposed system scales with the system size, especially when exceptions are frequent, unlike the conventional checkpoint-and-recovery method.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Parallel programming; D.3.4 [*Programming Languages*]: Processors - Run-time environments; D.4.5 [*Operating Systems*]: Reliability - Checkpoint/restart, Fault-tolerance

General Terms Design, Experimentation, Measurement, Performance, Reliability

Keywords Deterministic Multithreading, Precise Exceptions

1. Introduction

Evolving technology continually places new demands on computer system designers. As designers address challenges in one aspect of the system, they often impact another, and on occasion create new opportunities. This then evokes further response from them, perpetuating the cycle. Figure 1 summarizes some key recent trends. For example, in the past, when Moore's law and Dennard scaling improved processor performance, software automatically reaped the benefits. But once Dennard scaling slowed down, programmers could no longer assume continued improvements. Vendors responded to the Dennard scaling slowdown with multicores, requiring programmers to adapt with parallel programs.Deterministic execution was then proposed to simplify the resulting parallel programming challenges [6–8, 28, 35].

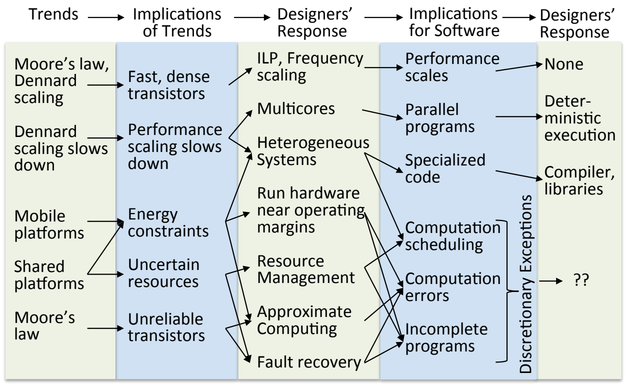


Figure 1. Impact of key technology trends on computer systems. Moore's law prevails, but Dennard scaling has slowed down. Hence, hardware no longer scales performance automatically, has multiple computational resources, is more unreliable, and is energy constrained. In response, programs need to be parallel, utilize resources efficiently, tolerate failures and conserve energy. Consequently, *discretionary exceptions* will likely be the norm in parallel programs.

Looking ahead, new techniques to conserve energy, improve performance, and manage resources are emerging. These techniques will give rise to *discretionary exceptions*, designer-permitted events that may alter the program's prescribed execution. As parallel programming proliferates, these trends will likely make frequent discretionary exceptions the norm in parallel programs. Nevertheless, designers will desire programs that execute as intended, as if an exception, especially a discretionary exception, never occurred.

Present approaches handle exceptions in a parallel program by periodically *checkpointing* its state. Upon exception, they *recover* to a prior error-free state and resume the program, losing all work completed since. A plethora of hardware [3, 34, 37, 43] and software [10, 11, 15, 27, 30, 39, 46] approaches, striking trade-offs between complexity and overheads, have been proposed in the literature (Table 1: rows 1, 2). Our qualitative analysis shows that their checkpointing and recovery processes will be too inefficient to handle frequent exceptions. In fact, they lack the scalability needed for future, increasingly larger systems. The question then arises, how will designers respond to the frequent discretionary exceptions in an efficient and programmer-oblivious manner?

In one response to this emerging challenge, we take inspiration from precise interruptible processors. Modern processors execute a sequential program's instructions in parallel, yet handle exceptions efficiently. They exploit the implicit order between the program's

PLDI '14, June 9 - 11 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06...$15.00.
http://dx.doi.org/10.1145/2594291.2594306

	Proposal	Recovery	Design	Chkpt. Cost	Rec. Cost	Scalable	Deterministic	Det. Cost
1	Rebound[3], ReViveI/O[34], ReVive[37], SafetyNet[43]	Yes	Hardware	High	High	No	No	N/A
2	[10, 11, 15], C^3[30], [39, 46]	User code	Software	High	High	No	No	N/A
3	DMP[13], RCDC[14], Calvin[24]	No	Hardware	N/A	N/A	N/A	Yes	High
4	dOS[7], CoreDet[6], Grace[8], DTHREADS[28], Kendo[35]	No	Software	N/A	N/A	N/A	Yes	High
5	**GPRS** (this work)	**Full program**	**Software**	**Low**	**Low**	**Yes**	**Yes**	**Low**

Table 1. Summary of related work. Some proposals (rows 1, 2) handle exceptions, but are not deterministic. They incur high checkpoint (Chkpt.) and recovery (Rec.) costs, and may not scale with the system size. Others (rows 3, 4) are deterministic, but do not handle exceptions, and can incur high costs (Det.). GPRS handles exceptions by making the execution deterministic, scalably and at lower overheads.

instructions to readily create a *consistent architectural state* when exceptions occur. This allows them to *precisely* recover from the exception and quickly restart the program with a minimum loss of work. Analogously, we propose *globally precise-restartable* execution of parallel programs. We impart a total order to a parallel program's computations and effect its *deterministic* execution, enabling low overhead and scalable exception handling.

Researchers have proposed several hardware and software techniques to execute a parallel program deterministically (Table 1: rows 3, 4). Although they list exception handling as one use of such an execution, they neither explore its implementation nor the related challenges. Of these, hardware proposals add considerable complexity [13, 14, 24]. Although software proposals [6–8, 28, 35] do not require special hardware, they may not handle arbitrary programs efficiently. We overcome these shortcomings in our work.

We draw from the past work on recovery and other areas, and introduce new features to build a *globally precise-restartable recovery system* (GPRS) for shared memory systems. Our goal is to preclude the need for complex hardware since future systems will rely primarily on software to handle exceptions [12, 32]. Hence we developed GPRS as a software runtime system. GPRS is also OS-agnostic. GPRS exploits program characteristics and user annotations to minimize overheads and achieve scalability. It incorporates the following novel aspects:

- GPRS applies the precise interrupt principles at the scale of multiprocessors. It introduces a notion of *selective restart*, in which not all computations, but only those that are affected are restarted, making GPRS scalable.
- GPRS exploits the synchronization points in data race-free programs to localize the impact of exceptions to a minimum number of program's computations.
- GPRS combines checkpointing and log-based approaches to minimize recovery overheads.
- GPRS achieves efficient deterministic execution by judiciously dividing program threads into *sub-threads* and employing a *balance-aware* schedule to execute them.
- GPRS is a fully-functional recovery system, operational on stock multiprocessors. It can handle exceptions in user code, third-party libraries, system calls, as well as itself.

We evaluated GPRS by applying it to recover from non-fatal exceptions in standard Pthreads parallel benchmarks. Experiments conducted on a 24 context multiprocessor system showed that GPRS outperformed the conventional checkpoint-and-recovery method. Importantly, it withstood frequent exceptions and scaled with the system size, whereas the conventional method did not, validating our qualitative analysis.

The paper is organized as follows. §2 examines sources of exceptions, the challenges and cost of processing them. It describes our model for globally precise-restartable execution. §3 presents the design and operational details of GPRS. Our experiments and the evaluation results are presented in §4. In §5 we put our work in the context of other related work, before concluding in §6.

2. A Case for Globally Precise-restartable Execution

2.1 Impact of Trends on Program Execution

As mobile and cloud platforms grow in popularity, and device sizes shrink, designers face energy and resource constrained, unreliable systems (Figure 1). Proposals being made to address these challenges can impact a program's execution, even terminally. Minimizing this impact will make these proposals practical. We analyze these proposals by dividing them into three broad categories.

Managing resources to save cost or energy can impact a program's execution. Shared systems, e.g., Amazon's EC2 [2] and mobile platforms [1], can abruptly terminate a program, even without notifying the user. Systems are now incorporating heterogeneous resources with disparate energy/performance profiles. To maximize benefits, dynamic scheduling of computations on these resources will be desirable [9]. One can also envision scheduling computations, even interrupting and re-scheduling currently executing computations, on the "best" available resource as the resources become available dynamically.

Further, emerging programming models can impact the program execution. Disciplined approximate computing [40], a recent research direction, permits hardware to inaccurately perform programmer-identified operations in a bid to save time and/or energy [17, 18]. Emerging proposals provide a software framework to compute approximately but with a guaranteed quality of service, by re-computing when errors are egregious [5]. In the future one can imagine integrating approximate hardware with such a software framework for even more benefits. Interestingly, some proposals tolerate hardware faults by admitting computation errors in a class of error-tolerant applications such as multimedia, data mining, etc., while ensuring that the programs run to completion [26].

Finally, hardware design can impact the execution. Faults due to soft (transient) errors, hard (permanent) errors and process variations can cause programs to crash or compute incorrectly on increasingly unreliable hardware [25].Hardware designers are employing techniques to manage energy consumption [22, 23] and device operations [45], which can lead to frequent timing, voltage and thermal emergencies. Like faults, these techniques can also affect the execution. Moreover, growing system sizes makes the systems more vulnerable to such vagaries.

In all of the above scenarios the program execution may be viewed as one interrupted by *discretionary exceptions*. These are exceptions that the system allows, in bargain for other benefits, such as energy savings. If these exceptions can be tolerated, efficiently, and oblivious to the programmer, designers can effect such and perhaps yet undiscovered tradeoffs. In fact, a similar trend ensued in uniprocessors when precise interruptibility provided a low overhead means to tolerate frequent exceptions. Once introduced, designers exploited the capability to permit high frequency discretionary exceptions arising from predictive techniques, for net gains in performance. For example, techniques to predict branch outcomes, memory dependences, cache hits, data values, etc., relied on precise interrupts to correct the execution when they mispre-

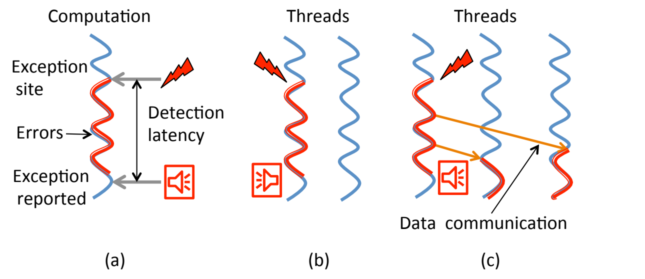


Figure 2. (a) Latency to detect and report an exception. (b) A *local* exception affects an individual computation. (c) A *global* exception affects multiple computations.

dicted. Hence we believe that recovering from exceptions in parallel programs and resuming the execution efficiently, and transparently, will be highly desirable in the future.

2.2 Exceptions

Recovering from exceptions is complicated by two factors. The first is the delay between the time an exception occurs and it is reported, which can range from one to many cycles (Figure 2(a)). For example, it may take tens of cycles to detect a voltage emergency [23], or an entire computation to detect errors in results [5]. The second, more pertinent to parallel programs, is due to the dispersal of a parallel program's state among multiple processors. To study the complexity, we categorize exceptions into two types: *local* and *global*. Local exceptions, e.g., page faults, impact only an individual thread, and can be handled locally without affecting other concurrent threads (Figure 2(b)). Local exceptions are often handled by using precise interrupts in modern processors [23].Global exceptions, in contrast, may impact multiple threads simultaneously, e.g., a hardware fault that corrupts a thread whose results are consumed by other threads before the fault is reported (Figure 2(c)). Due to the inter-thread communication in parallel programs and the system-wide impact of global exceptions, global exceptions are complex to handle, and hence are this work's main focus.

2.3 Recovery from Global Exceptions

The traditional method to recover from a global exception is to periodically checkpoint the program's state during its execution, often on stable storage. Upon exception, the most recent possible, error-free consistent architectural state is constructed from the checkpoint, effectively rolling back the execution. The program is then resumed from this point. There are three main types of such *checkpoint-and-recovery* (CPR) methods: *coordinated*, *uncoordinated*, and *quasi-synchronous* [16, 29]. Of the three, coordinated CPR has gained popularity in shared memory systems since it is relatively simple and guarantees recovery at relatively lower overheads. Hence we consider coordinated CPR here.

Parallel programs are characterized by nondeterministic concurrent threads that may communicate with each other. Hence, the execution needs to be quiesced to take a checkpoint of the state that is consistent with a legal execution schedule of the program up to the checkpoint [16]. A consistent checkpoint is essential for the program to restart and execute correctly. The lost opportunity to perform work, i.e., the loss of parallelism, during the CPR processes effectively penalizes the performance. Hence, CPR schemes incur a *checkpoint penalty*, P_c, every time they checkpoint, and a *restart penalty*, P_r, when they restart from an exception.

To checkpoint, prevailing software approaches [10, 11, 15, 30, 39, 46] first impose a barrier on all threads, and then record their states (Figure 3(a)). A second barrier is used to ensure that a thread continues the execution only after all others have checkpointed, to prevent the states they are recording from being modified. The checkpoint penalty (P_c) is proportional to the average time each context spends coordinating at the two barriers, t_c, recording its state, t_s, and the checkpointing frequency. Ignoring actual mechanisms and assuming contexts can record concurrently, in an n context system, for a checkpoint interval of t sec, $P_c = \frac{1}{t} \cdot n \cdot (t_c + t_s)$.

To restart an excepted program, the program is stopped and the last checkpoint is restored (Figure 3(b)). In this case the potential loss of parallelism arises from the work lost since the last checkpoint, i.e., over the interval t, and from the wait time, t_w, to restore the state. Therefore, the total *restart delay*, $t_r = t + t_w$, and for a rate of e exceptions/sec, the restart penalty $P_r = n \cdot e \cdot t_r$.

Recent hardware proposals can reduce the two penalties by involving only those threads (n_c) that communicated with each other during a given checkpoint interval, in the checkpoint and recovery processes [3, 37]. Hence in their case, $P_c = \frac{1}{t} \cdot n_c \cdot (t_c + \frac{n}{n_c} \cdot t_s)$ (all threads still record the state), and $P_r = n_c \cdot e \cdot t_r$.

We believe that as exceptions become frequent, the restart penalty will become critical. Intuitively, if exceptions occur at a rate faster than checkpoints, the program will never complete. Hence, for a program to successfully complete, it is essential that $n \cdot e \cdot t_r \leq n$, i.e., $e \leq \frac{1}{t_r}$. The hardware proposals can improve this to $e \leq \frac{n}{n_c} \cdot \frac{1}{t_r}$. Decreasing P_r by simply increasing the checkpoint frequency, i.e., decreasing t, will increase P_c, and hence may be unsuitable for high frequency discretionary exceptions.

We believe that an online system that can quickly repair the affected program state, with minimum impact on the unaffected parts of the program, analogous to the precise interrupts in microprocessors, will be needed when exceptions are frequent. Hence, for a practical and efficient solution, all factors of checkpoint and restart penalties need to be reduced.

2.4 An Efficient Approach to Frequent Exception Recovery

To build a practical and efficient exception recovery system for frequent discretionary exceptions we target all the related overheads: the discarded work, the checkpoint frequency, the coordination time, the recording time, and the restart process. For this purpose we formulate a *globally-precise* exception model. We present the model, its key design aspects, and overhead tradeoffs here. In the next section we describe an actual system design based on it.

To formulate the globally-precise exception model, we take a fundamentally different view of the parallel program execution, and make three logical changes.

1. Order Computations. Ignoring threads for the moment, ideally, when an exception occurs, only the affected computations, i.e., the excepted computation and those which may have consumed the erroneous results produced by it, should be squashed and restarted. For example, in Figure 3(c), if only computation C consumes the data produced by B, when B excepts only B and C need restart, without affecting A. Parallel programs permit arbitrary and nondeterministic communication between computations, making it difficult to identify the precise dynamic producer-consumer relationships between them. Hence CPR schemes take the conservative approach of discarding all computations after an exception.

To overcome the limitations of nondeterminism we impart a logical order to the program's computations, although they may execute concurrently (Figure 3(d)), an approach similar to recent proposals on deterministic execution [6, 8, 28, 35]. The implicit order precludes communication from "younger" computations to "older" computations. Hence, an affected computation cannot corrupt older computations. When an exception occurs, undoing the effects of the excepted computation and all younger to it results in a program state that is consistent with the execution. The state is "precise", reflecting execution of all older computations and none

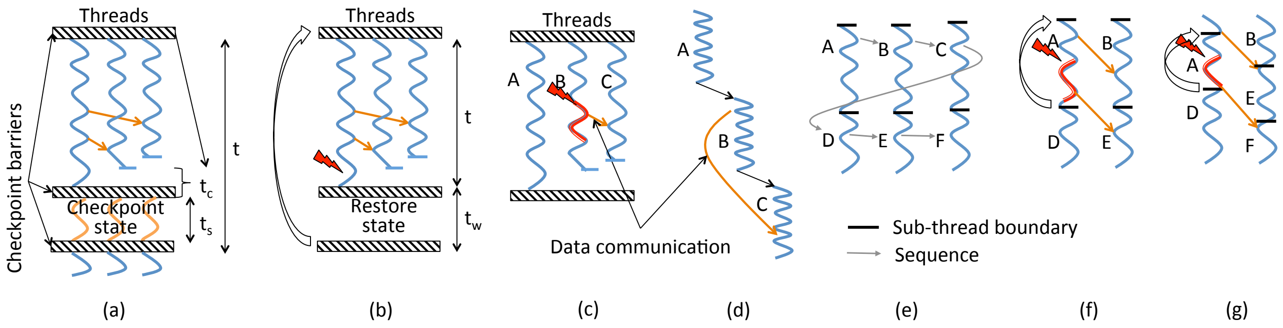


Figure 3. (a) Conventional checkpoint-and-recovery. (b) Restart penalty. (c) Inter-thread communication. (d) Ordered computations. (e) Creating and ordering sub-threads. (f) Inefficient sub-thread boundaries. (g) Sub-thread boundaries at communication points.

of the younger ones, analogous to the state a precise-interruptible processor creates upon exception. Hence we term such an execution, *globally precise-restartable*.

For exception recovery, a globally precise-restartable execution can reduce the restart penalty, as compared to CPR, since not all computations, but only the excepted and younger computations need be discarded. The ordering further enables the desired *selective restart* of only the affected producer-consumer computations, and not of other unrelated older or younger computations. This minimizes the amount of discarded work. This is also analogous to re-executing only an excepted instruction and its dependent instructions in superscalar processors, e.g., a load that misses in the cache but was presumed to have hit, without squashing all in-flight instructions.

Selective restart potentially reduces the restart penalty to $P_r = e \cdot t_r$ (assuming that the average computation size is t and time t_w is taken to reinstate its state), since only the excepted computation may need to restart and the unaffected computations need not stall. This also enables an online system that can continuously respond to frequent discretionary exceptions while minimizing the impact on the program's unaffected parts. For a program to now successfully complete, $e \cdot t_r \leq n$, i.e., $e \leq \frac{n}{t_r}$. Thus selective restart is potentially $n\times$ more exception-tolerant than the conventional CPR ($e \leq \frac{1}{t}$), and $n_c\times$ more tolerant than the recent hardware proposals ($e \leq \frac{n}{n_c} \cdot \frac{1}{t_r}$), making it more effective in parallel systems.

An alternative to imposing an order on computations is to permit the nondeterministic execution but record the dynamic order between the computations at run time. The recorded order can also be used to perform the selective restart. The choice between deterministic and nondeterministic approaches has implications on the implementation and the actual use. For example, nondeterministic order may complicate the design when exceptions occur in the implementation mechanisms (§3.2). In another example, deterministic order may give the desired repeatability more readily than nondeterministic order for memoization-based approximate computing. On the other hand, deterministic order can degrade performance (§3.2). Still, deterministic ordering has broader applicability, as is noted by others [8, 35]. Hence, in this work we implement deterministic ordering, but overcome the performance issue it poses (§3.2). We explore techniques for the two approaches and the related tradeoffs in future work.

2. Create Sub-threads. Next, we apply an order to the threads. A naive approach could order the threads themselves, which may serialize the execution. Moreover, the restart penalty will be too large, given the granularity of threads. Hence we logically divide the threads into finer-grained sub-threads. We then view the sub-threads to be ordered across the threads, effectively creating a total order. Figure 3(e) shows one example of how threads may be divided into sub-threads, with a round-robin order, A-B-C-D-E-F. Note that this order is one legal schedule of a parallel program's execution. This prevents the serialization of the execution and the impact on the restart penalty.

The sub-threads are logically created at communication points in the threads, thus restricting the communication to sub-thread boundaries. Checkpoints are taken at the start of sub-threads. This yields two benefits. First, it helps to localize an exception's impact. For example, in Figure 3(f), an exception in sub-thread A forces restart of B and E, whether they consumed A's erroneous data or not. By restructuring the boundaries, as shown in Figure 3(g), only the newly created F need restart, localizing the impact.

Finer-grained sub-threads increase the checkpoint frequency (reduce t), further reducing the restart penalty, P_r, but increase the checkpoint penalty, P_c. The second advantage of checkpointing at the sub-thread boundary is that at that time no other sub-thread can be communicating with it. This eliminates the need to coordinate (t_c), entirely, reducing P_c to $\frac{1}{t} \cdot n \cdot t_s$ (average sub-thread size is t).

Although the model reduces the checkpoint and restart penalties, it introduces a new penalty when applying the deterministic order and housekeeping for selective restart. When a context enforces order on a sub-thread and checkpoints its state, it can delay the sub-thread's actual execution (to be seen in §3). If the average delay is t_g, the total penalty of the globally precise-restartable model, $P_g = \frac{1}{t} \cdot n \cdot t_g$. As we show in the evaluation, the overall benefits of the model far outweigh this penalty.

Incidentally, creating fine-grained sub-threads allows us to employ a task-style scheduler to balance the load in the system, yielding orthogonal performance benefits.

A key aspect of the model is to identify the data communication points in the program. The model's present implementation assumes that programs use standard APIs and are data race-free, or contain user annotated data race-prone regions. We discuss this aspect further in §3.

3. Perform Application-level Checkpointing. Finally, we reduce the recording overhead, t_s, by resorting to application-level checkpointing, which can dramatically reduce the checkpoint sizes [10]. Instead of taking a brute-force checkpoint of the system's entire state [27] or the entire program [15, 39], only the state needed for the program's progress is recorded. Often, instead of saving data, data can be easily re-computed, especially if the computation is idempotent [38]. Several compiler proposals automate application-level checkpointing [11, 30]. Alternatively, the user's intimate knowledge of the program can be exploited for this purpose by requiring the user to annotate the program. Our current implementation takes the latter approach.

While the above principles enable a responsive and efficient recovery model for discretionary exceptions, it presents several

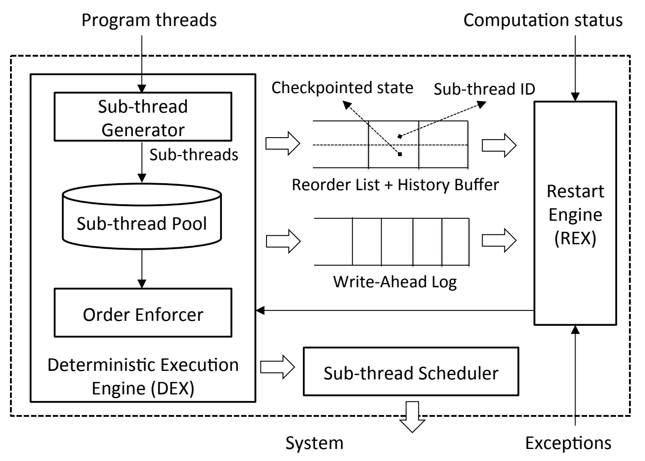


Figure 4. GPRS System Architecture.

design challenges. We next describe these challenges and how one embodiment of the model, GPRS, addresses them while automating the recovery mechanisms.

3. GPRS Design and Implementation

The globally precise-restartable recovery system, GPRS, is implemented as a C/C++ run-time library. The library currently works with the Pthreads APIs and `gcc/g++` interfaces for atomic operations, although it can be easily extended to other APIs. We present an overview of the GPRS architecture, followed by its design details. GPRS is an extensive system and presents many design choices. We visit some key choices in the description.

3.1 Architecture Overview

Figure 4 shows the block diagram of GPRS. GPRS interposes between a parallel program and the underlying system. It manages the program's execution, the program's state, and shepherds the execution to completion when exceptions, discretionary or otherwise, arise during the program or from the system itself.

GPRS comprises two main components: the Deterministic Execution Engine (DEX), and the Restart Engine (REX). The DEX is analogous to the out-of-order execution engine in superscalar processors. It intercepts the program at run time and creates the sub-threads, unbeknownst to the programmer. It orders the sub-threads, records their state and schedules their execution. Trivially ordering the sub-threads can severely impact the program's performance, as we show below. DEX uses a novel scheme that respects the program's parallelism pattern to minimize this impact.

Discretionary exceptions can not only affect the user code and functions invoked from it, but also GPRS's own mechanisms. For example, a fault due to voltage emergency can corrupt GPRS operations, which can ultimately corrupt the user program. GPRS is uniquely capable of handling exceptions in its own operations. For this purpose GPRS records its operations in a log, instead of checkpointing its state. It leverages the sub-thread ordering to optimize this process, another novel aspect of GPRS.

The REX tracks the completion of sub-threads and responds when exceptions arise. REX is analogous to the misprediction and precise exception handling logic in microprocessors. It reconstructs the program's error-free state and that of the GPRS, using the recorded state and the operation log. It then instructs the DEX to resume the execution from this reconstructed "safe" point.

The DEX communicates with the REX through a reorder list and a history buffer, which holds the recorded state and sub-thread

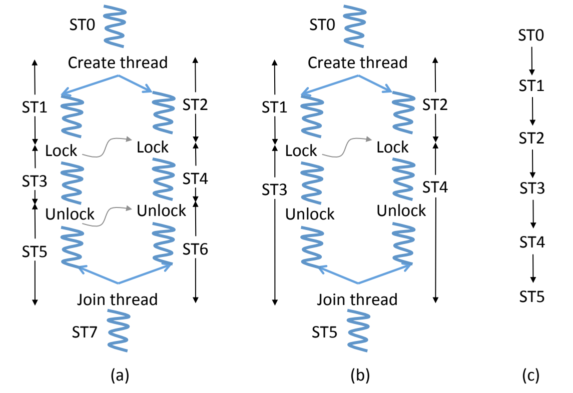


Figure 5. (a) Creating and ordering sub-threads. (b) Optimized sub-threads. (c) Sub-thread order.

information. GPRS also uses a load-balancing scheduler, popularized by others [19], to farm the sub-threads for execution.

3.2 Deterministic Execution Engine (DEX)

The DEX intercepts all dynamic instances of Pthreads API calls and `gcc/g++` atomic operations invoked from the user program. It replaces each call type with its own logic, much like other deterministic execution proposals [28, 35], to create and manage the ordered sub-threads. Currently GPRS does not support non-standard, home-spun atomic operations, which are otherwise discouraged practices. Nonetheless, GPRS provides a conservative alternative to handle them, as described later.

Creating Sub-threads. As the program executes, a *sub-thread generator* creates a pool of sub-threads (Figure 4). The start of the program initiates the first sub-thread. Subsequent calls to create threads, terminate threads, atomic operations, and invoke critical sections cause the sub-thread generator to logically divide the threads into sub-threads. In general, each such call signifies the end of the preceding sub-thread and the start of new sub-thread(s). For example, as Figure 5(a) shows, when `pthread_create` is invoked, the sub-thread generator terminates the parent sub-thread ST0, creates a sub-thread ST1 from the parent thread, and creates another sub-thread ST1 from the newly forked thread. When ST1 invokes a critical section, e.g., by using `pthread_mutex_lock`, the sub-thread generator terminates ST1 and creates a new sub-thread, ST3, starting from the call. Appropriate actions are similarly performed for the other API calls. Note that a sub-thread is not the same as an OS thread, although both maintain similar data structures, e.g., stacks.

Our experiments showed that the critical sections in programs are typically very small. Hence the benefit of creating sub-threads at both Pthreads `lock` and `unlock` points, e.g., ST3 and ST5 in Figure 5(a), did not offset the ordering overhead. Hence we optimized the design to not create a new sub-thread at the `unlock` points. The critical section and the succeeding code are assigned to the same sub-thread, e.g., ST3 and ST5 in Figure 5(a) are assigned to ST3 in Figure 5(b). We also logically flattened nested critical sections into the outermost critical section. The sub-thread generator tracks the `lock` and `unlock` calls. When a `lock` is encountered before an `unlock` matching the preceding `lock`, it is subsumed in the enclosing critical section; no new sub-thread is created.

Other logical sub-thread divisions are also possible, for e.g., based on the dynamic instruction count [6, 35]. We found our scheme to be simpler and adequate.

Ordering Sub-threads. Once created, an *order enforcer* (Figure 4) assigns a deterministic order to the sub-threads and submits them

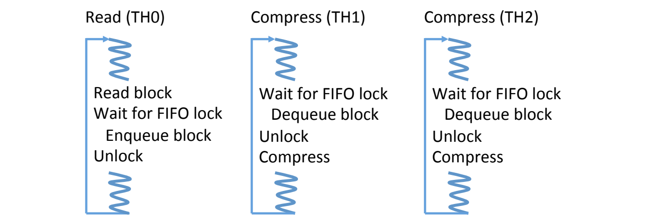


Figure 6. Logical view of Pbzip2's read and compress threads.

for execution in that order. To order the sub-threads, a simple scheme would be the round-robin order, based on the order of each sub-thread's parent thread in the program text, e.g., as shown in Figure 5(c) for the sub-threads in Figure 5(b). If a sub-thread arrives at a synchronization point before the logically preceding sub-threads arrive at their synchronization points, the sub-thread will pause until it is its turn to proceed. DTHREADS uses a similar scheme by passing a global token across the threads at the synchronization points [28]. Such an order, however, can severely impact the program's performance.

Often programmers carefully orchestrate a parallel program's execution to maximize performance. A common technique is to distribute work evenly among the threads. More complex techniques keep threads busy by ensuring that the threads obtain data with minimum delay. A simple round-robin schedule can neutralize such techniques.

For example, consider the popular high performance implementation of Pbzip2 [20]. It reads data blocks from an input file, one at a time, compresses them concurrently, and writes the results to an output file, creating a logical pipeline between the three types of operations. It launches one read thread, multiple compress threads, and one write thread for balanced work distribution. The read thread communicates data to the compress threads through a lock-protected FIFO. The compress threads communicate similarly with the write thread via another FIFO. Figure 6 shows the read (TH0) and compress (TH1, TH2) sub-threads that execute iteratively (the write thread in not shown). They access the FIFO in a critical section implemented using conditional wait-signaling.

Figure 7(a) shows Pbzip2's execution, in time epochs, using the round-robin order, TH0-TH1-TH2. Recall that the order is being enforced at the FIFO access critical section. In epoch t0, TH0 reads a block from the file, and since it is its turn, it enqueues the block in the FIFO, and passes the global token to TH1. Next, in epoch t1, TH1 accesses the FIFO, dequeues the block, passes the token to TH2, and begins work on the block in t2. Meanwhile, in t1 TH0 reads the next file block, which it can enqueue in the FIFO only when its turn arrives next. In epoch t2, TH2 accesses the FIFO, but finds it empty, and passes the token back to TH0. TH0 now puts the block in the FIFO in t3, and passes the token to TH1. TH1, however will access the FIFO only when it completes its current work, in t5, at which point it will dequeue the block, and pass the token to TH2. Meanwhile, TH2, waits for its turn at the FIFO, and when it gets the turn, in t6, it will once again find the FIFO empty. This process repeats, in which TH0 reads a block that only TH1 can access, starving all other compress threads, essentially serializing the execution. A similar construct exists between the compress threads and the write thread, leading to a similar bottleneck. Thus the round-robin order will be grossly inefficient. Kendo [35] and CoreDet [6] use a slightly different scheme, but also attempt to give all threads equal opportunity, resulting in unused cycles.

The above scenario arises because the round-robin scheme defeats the program's original execution plan by introducing an artificial order. It dissolves the parallel pipeline structure, and by regimenting the communication, creates an imbalance in the work division among the threads. To overcome the artificial order limitations we use a *balance aware* ordering schedule, analogous to dataflow execution used in out-of-order processors to overcome the sequential order limitations. A formal approach to automatically find the best balance-aware schedule is a subject of our ongoing research. We present two schemes, *basic* and *weighted*, currently implemented in the DEX.

The DEX divides the program's threads into *thread groups*, where each group represents a computation type. For example, in the above case, the read thread is assigned to one group, the compress threads to another, and the write thread to yet another. The DEX assigns order to the threads hierarchically. First, a round-robin schedule is used for threads within a group, since all threads perform the same type of computation. Next, across the groups, an order reflecting the programmer-enforced balance and communication structure, is assigned. In the basic scheme, the DEX assigns a uniform round-robin order to the groups.

Figure 7(b) shows Pbzip2's execution using the basic balance-aware order (ignore the write thread). In epoch t0, group 0, and within group 0, TH0 receives the token. It enqueues the block in the FIFO, and passes the token to group 1 in t1. Within group 1, TH1 receives the token, upon which it dequeues the block, passes the token back to group 0 and not TH2, and begins its own work. At t2, TH0 in group 0, being the only context in the group, receives the token, enqueues another block in the FIFO, and passes the token to group 1. In t3, this time, TH2 and not TH1 receives the token, upon which it dequeues the block from FIFO and receives the work to perform. This process then repeats, permitting all stages of the program's original pipeline structure to proceed concurrently, analogous to dataflow execution, overcoming the artificial ordering.

In the more advanced, weighted scheme, threads within a group receive the round-robin order, but across the groups, critical groups are given more "weight". Threads in groups with higher weight receive more turns at the execution. For example, the early stages of a pipeline can be weighted higher so that they can produce enough data to keep the downstream stages busy. In the case of Pbzip2 (not shown), TH0 may receive the token multiple times before it is passed to TH1 and TH2, to achieve an effect similar to Figure 7(b).

Thus, the balance-aware schemes provide mechanisms to restore the program's original workload balance and dataflow.

Although compiler techniques may be used to identify the parallelism patterns [21], GPRS currently takes the input from the programmer. The `pthread_create` API was extended to pass a group ID to which the new thread is assigned. This input is enough for the basic scheme. To employ the weighted scheme, the group's weight can also be specified through the API.

Managing the Program State. Once the DEX assigns an order to a sub-thread, it inserts an entry for the sub-thread into the reorder list (ROL), signifying its order among the currently in-flight sub-threads (Figure 4). The ROL is analogous to a reorder buffer (ROB) in superscalar processors. Before a sub-thread is finally submitted for execution, the DEX records the state necessary to restart the sub-thread. It checkpoints the sub-thread's call stack, the processor registers, and any data the sub-thread may modify, i.e., its *mod set*. This state is maintained in a history buffer (Figure 4). GPRS also logs the dynamic identity of any lock(s) the sub-thread may have acquired or the atomic variable it may have accessed, as an alias for the shared data the sub-thread accesses. This information is used during recovery (§3.4).

Several approaches may be taken to identify a sub-thread's mod set. Compiler techniques [10], or copy-on-write scheme may be used. GPRS presently requires the programmer to provide the checkpointing functions through the Pthreads APIs.

Third Party, I/O, and OS Functions. User programs often comprise third party, I/O and system functions. These functions pose

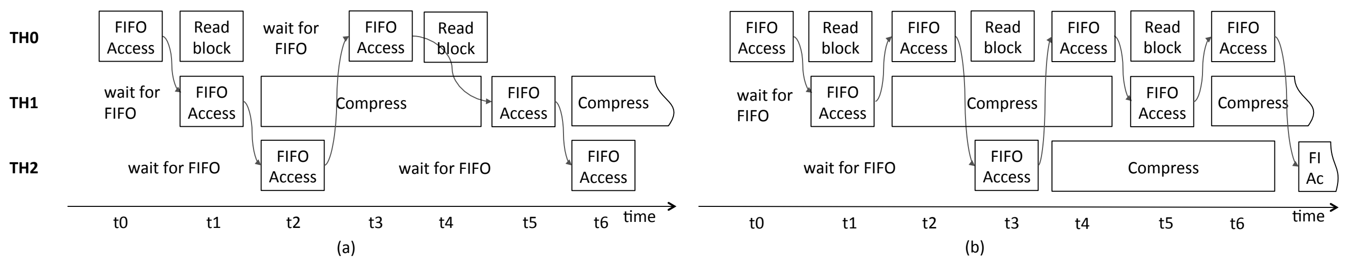


Figure 7. Pbzip2 execution. (a) Round-robin order serializes the execution. (b) Balance-aware order restores the parallelism.

unique challenges to recovery and restart since GPRS must have access to their mod sets and the ability to re-execute them, which may not always be the case.

Since GPRS uses application-level checkpointing, if the user provides the checkpointing functions for the third party and system functions, they can be handled just like user functions. If a function's mod set is unknown and the user identifies the function to GPRS, GPRS can strictly serialize its execution by ensuring that all preceding sub-threads complete execution before it, and no subsequent sub-threads execute concurrently, relying on local recovery if an exception occurs.

To widen its utility, GPRS implements its own version of the more commonly used system functions, the memory allocator and the file I/O calls. It automatically handles exceptions (described below) in these functions, with no further input from the user.

Restarting I/O operations can pose a challenge if they need to be re-performed. A certain class of I/O operations, e.g., file read and writes, can be made idempotent [41], which GPRS does. Such operations can be simply re-executed. For non-idempotent operations, e.g., network I/O, GPRS resorts to sequential execution, similar to handling functions with unknown mod sets.

The output-commit problem, which requires that only non-erroneous data be committed to the (non-idempotent) output, is overcome by waiting for the exception-detection latency before committing the data to output [42].

Managing the Runtime State. The GPRS runtime employs data structures and sophisticated concurrency algorithms in its own operations. These data structures include work queues, lock queues, memory allocator lists, the ROL, and other book-keeping structures. The concurrent algorithms operate at very fine granularities. Applying conventional CPR to the runtime will lead to the same problem that it is attempting to solve. Instead we take advantage of the created order, and a write-ahead-logging (WAL) technique, inspired by the Aries recovery mechanism of DBMS [33], to recover the internal state of GPRS (§3.4).

We note that each runtime operation is performed on behalf of a sub-thread. Hence we assign the operation the associated sub-thread's order. Before operating on a runtime structure, each GPRS context independently logs the operation along with its unique ID, to the WAL, which is maintained on error-free stable storage.

3.3 Load-balancing Scheduler

The DEX submits the sub-threads for execution to a load balancing *sub-thread scheduler* (Figure 4). When a program begins execution the scheduler creates a pool of OS threads, one per hardware context. These threads actively seek work, minimizing the idle time. Sub-threads form a unit of work for these threads, and due to their granularity lend themselves well to the load-balancing function.

3.4 Restart Engine (REX)

The REX plays two roles in GPRS. It tracks the sub-threads' execution and recovers from exceptions.

Retiring Sub-threads. GPRS maintains all the information needed to rollback and restart a sub-thread only as long as it is needed, after which it *retires* the sub-thread. A sub-thread can retire when, (i) its execution completes without raising any exceptions, and (ii) it is the oldest in-flight sub-thread, i.e., an exception in another sub-thread cannot cause its rollback. When a sub-thread completes without exception, the REX records its status in its corresponding ROL entry. The REX continually monitors the ROL head, which holds the entry of the oldest sub-thread not yet retired. When the ROL-head entry completes exception-free, the corresponding sub-thread is retired by removing the entry from the ROL and deleting the sub-thread's checkpointed state.

Precise Recovery from Exceptions.

REX provides several exception recovery options depending on the use case and system capabilities. REX can recover precisely from the sub-thread boundary, or if the exception-detection latency permits, precisely from the instruction boundary. It can also perform hybrid recovery, combining selective restart and CPR, for programs not written suitably for the model. The REX recovery code typically executes on the excepted processor, like an interrupt service routine. We describe the basic recovery mechanism and build on it to describe selective restart and hybrid recovery.

Basic Recovery. In the basic recovery, REX first constructs the precise architectural state. When a sub-thread excepts, the REX halts its execution, records its status in its ROL entry and waits for it to reach the ROL head. During its ROL monitoring, when the REX detects an excepted entry at the head, it temporarily pauses the program by halting all executing sub-threads. It restores the architectural state modified by the younger sub-threads, using their checkpointed state, in the reverse ROL order, effectively squashing their execution. This constructs the program's precise architectural state up to the excepted sub-thread.

The REX then restores the GPRS data structures using the WAL logs. In general, to restore the runtime's precise state, GPRS walks the logs in the reverse order and undoes the operations performed for the squashed sub-threads. This restores the data structures to reflect the operations performed up to the excepting sub-thread. The logs are pruned as the sub-threads retire to keep their sizes bounded.

If the system can raise instruction-precise exceptions, REX relies on precise-interruptible microprocessors to create the precise architectural state. Since a sub-thread executes as a unit on a processor, the processor executing the excepted sub-thread contains the precise state up to the excepted instruction. With the state of other sub-threads unrolled as above, the system now reflects instruction-precise execution of the program, and may be restarted from here.

If the detection latency makes the exception imprecise, REX can perform sub-thread-precise restart, by squashing the excepting sub-thread in addition to the above rollback.

If the precise excepting sub-thread cannot be identified for any reason, it is always safe to discard all sub-threads in the ROL, undo their modifications, and restart the execution from the ROL head.

Selective Restart. To achieve selective restart, the basic recovery scheme is modified slightly. When a sub-thread excepts, the REX pauses the program's execution without waiting for the sub-thread to reach the ROL head. It walks the ROL to identify younger "dependent" sub-threads, ones that acquired the same lock(s) or used the same atomic variable as the excepting sub-thread. It then restores the architectural state modified by only the affected sub-threads, and restores any affected internal data structures. Once the state is restored, all unaffected sub-threads resume while the affected sub-threads restart. Thus selective restart is achieved.

Hybrid Recovery. If a program contains data races or uses non-standard APIs in some sections, GPRS can employ a hybrid recovery scheme. GPRS can apply CPR in such a section and selective restart in other parts of the program. To apply CPR, GPRS relies on the user to enclose the section between GPRS provided `start_cpr` and `end_cpr` calls, and provide a checkpointing function for the section through `start_cpr`. During the execution, when a `start_cpr` is invoked, the ensuing execution is treated as a sub-thread and any further sub-threading is stopped. A subsequent `end_cpr` call signals the end of such a section, and sub-thread creation resumes. Exception recovery remains the same as before.

3.5 Other Design Issues

Data Races. Since GPRS relies on standard APIs, it requires programs to be data race-free. Data races are problematic, in general. Hence emerging programming language standards, e.g., C++ and Java, are advocating date race-free programs. In fact, we consider data races to be exceptions. Many tools can detect data races. GPRS can be combined with an exception-raising race detector [44] to avoid data races altogether in a program execution. The GPRS scheduling policy can be changed to execute the detected race-prone sub-threads sequentially. We are currently exploring hardware/software techniques to automatically handle data races in GPRS.

Other Limitations. Presently, GPRS cannot support transient external events, e.g., interrupts and signals, during restart, although they can be handled by using methods others have identified [42].

Other Applications. Although here we have considered recovery from non-fatal exceptions, GPRS can be easily extended to also handle fatal exceptions such as permanent hardware faults.

4. Experimental Evaluation

To evaluate our approach we tested GPRS using standard parallel benchmark programs running on a stock multiprocessor system. We "raised" exceptions during program runs and applied GPRS to handle them. We assessed the overheads introduced by the GPRS mechanisms, its checkpointing and restart penalties, its effectiveness in handling exceptions, and its scalability, as compared with the conventional CPR method.

To exercise the different aspects of GPRS we chose a mix of programs that exhibit different characteristics: computation sizes, use of synchronization primitives, and critical section sizes (columns 2-4, in Table 2). Critical sections in the standard parallel benchmarks are typically small. Hence we included RE [4], a network packet processing program, which uses relatively larger critical sections. Pthreads versions of the programs were used. They were compiled using gcc 4.6.1, with the `-O3` and `march=corei7-avx` options. Experiments were conducted on a 2-way hyperthreaded, 6-core, two-socket (total of 24 contexts) Intel Xeon E5-2420 (Sandy Bridge; 32KB each, L1 I and D, 256KB private L2 caches per core;

Programs (1)	Comp. size (2)	Sync. op. freq. (3)	Crit. size (4)	Exec. time (s) (5)	Sub-thread size (6)	# Sub-threads (7)
Barnes-Hut	Large	Low	N/A	41.70	Med.	75076
Blackscholes	Large	Low	N/A	112.89	Med.	100002
Canneal	Small	Med.	Small	6.93	Small	6272
Swaptions	Large	Low	N/A	57.27	Large	130
Histogram	Small	Low	N/A	0.22	Small	26
Pbzip2	Med.	High	Small	17.89	Med.	42269
Dedup	Small	High	Small	73.71	Small	1377855
RE	Med.	Med.	Med.	7.70	Med.	102
WordCount	Small	Low	N/A	1.44	Small	54
ReverseIndex	Small	Med.	Small	3.37	Small	78430

Table 2. Programs and their relative characteristics. (1) Benchmark; (2) Default computation size; (3) Frequency of synchronization operations; (4) Critical section size; (5) Pthreads execution time for large inputs on 24 contexts; (6) Sub-thread size in GPRS; (7) Total fine-grained sub-threads created by GPRS.

15MB shared L3 cache) machine. It ran the Linux 2.6.32 kernel. We report execution times and overheads based on the wall-clock time obtained from linux for the entire programs. Large inputs were used for all programs. Results shown include any file and other I/O the programs perform, and are averages over ten runs. Column 5 in Table 2 gives the baseline, 24-thread Pthreads execution time.

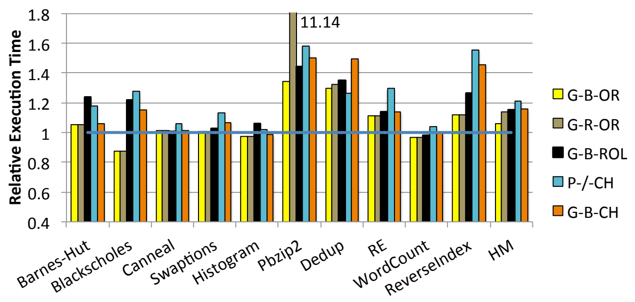
System Assumptions. We assumed that mechanism(s) exist in the system to raise exceptions. In an approximate computing framework, this could simply be a call from the user code to GPRS. In a fault tolerance system, the system may detect hardware faults using one of many prevailing techniques [31, 42] and report the excepted context to GPRS. Irrespective of the use case, we emulated this by launching an additional thread in the programs. The thread uses Pthreads signals to periodically signal GPRS and randomly designate one hardware context as excepted. We conservatively assumed an exception detection latency of 400,000 cycles (as have others [43]) to amplify the GPRS overheads. We also assumed stable storage is available to maintain the runtime logs. Recovery from a corrupt stable-storage, exceptions that cannot be attributed to a context, and system-level policies, e.g., handling permanent, repeating or non-recoverable failures are beyond this work's scope. We stress-tested GPRS under various exception rates, without emphasizing the probability distribution of the exceptions.

Program Sub-threads. All the programs we used take the number of system contexts as an input and fork as many threads. By default, Barnes-Hut, Blackscholes, Swaptions, Histogram, and Word Count divide work equally among data-parallel threads. For these programs GPRS treated each forked thread as a sub-thread. Pbzip2, Dedup, RE, and Canneal use critical sections in their threads. GPRS created sub-threads at their synchronization points (fork, join, lock, unlock, barrier, and signal-waiting operations). Reverse Index uses both, threads with critical sections and data-parallel threads; GPRS created the sub-threads accordingly.

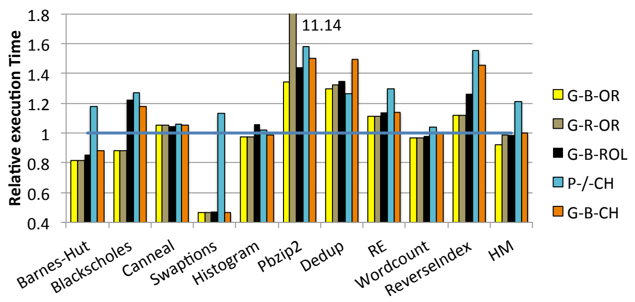
GPRS Overheads.

Result 1. *Balance-aware GPRS imposes modest basic overheads, 15.49% on average (harmonic), but up to 44.03% in case of Pbzip2 due to its large number of synchronization operations and tasks of uneven sizes.*

Figure 8(a) highlights the basic GPRS overheads ($P_g = \frac{1}{t} \cdot n \cdot t_g$, t = checkpointing interval, n = number of contexts, t_g = ordering+ROL management costs). It shows execution time of programs relative to the Pthreads baseline (marked by the horizontal line). The programs in this graph use default computation sizes, resulting in coarse-grain sub-threads, which we use to study the basic overheads before using the actual fine-grain sub-threads. The graph



(a) GPRS overheads using default computation sizes.



(b) GPRS overheads using finer grained computations.

Figure 8. GPRS overheads. Legend: System-Ordering-Overhead: System (G = GPRS, P = Pthreads); Ordering (B = balance-aware, R = round-robin, / = none); Overhead (OR = ordering, ROL = OR+ROL management, CH = OR+ROL+checkpointing). HM = Harmonic mean.

shows the cumulative overheads from the successive GPRS mechanisms: ordering (round-robin and balance-aware) and ROL management, and from the checkpoint penalty.

The overheads of enforcing order between a program's threads depend on the ordering policy, its frequency, and the time spent waiting for turns. In general, the round-robin ordering overhead (G-R-OR bars) is negligible for all programs except Pbzip2, Dedup, RE, and Reverse Index. The other six programs simply fork and join threads. The threads already have to wait at join points, making it convenient to create and order the sub-threads at the same time, at a small additional cost, if any. However, when the order is applied at the synchronization points in addition to the fork and join points, the load imbalance between the sub-threads penalizes performance, e.g., 10.9% in the case of RE and 12.1% for Reverse Index.

The round-robin order severely degrades Pbzip2's performance. As discussed in §3.2, it serializes the execution, resulting in an overhead of 1014.4%. When the basic balance-aware schedule was applied to Pbzip2, by passing the thread-group information to GPRS, the overhead dropped to 34.14% (G-B-OR bars). Pbzip2 comprises a large number of uneven-sized sub-threads and hence its relatively higher overheads. Applying the advanced schedule, by weighting the read, compress, and write thread-groups in the ratio 4:4:1, further reduced the overhead to 11.22% (not shown). This ratio was found by trial and error.

Dedup, another data compression program, is similar to Pbzip2 in its pipeline structure, but exhibits different characteristics. It uses five pipeline stages. The first reads a large block of data from a file. From this single block, the second stage creates smaller chunks that the following two stages operate on in parallel. Dedup's execution is dominated by the fifth sequential stage (file output), which remains the only running thread for the majority of the duration. Hence Dedup scales poorly. Further, GPRS creates $\tilde{1}$.38M Dedup sub-threads, resulting in a 32.16% overhead (G-R-OR). Despite the large number of sub-threads the overheads are not higher because the very fine-grained sub-threads do not create an ordering imbalance, unlike in Pbzip2. The basic balance-aware schedule only marginally reduced the overhead to 29.7% (G-B-OR). Other schedules did not fare better. Since one sub-thread from the first stage can create work for multiple downstream threads, and the individuals sub-threads are too small, the round-robin schedule did not create an imbalance in the first place.

On average (harmonic), the round-robin schedule introduces 14.01% overhead (HM G-R-OR) and the basic balance-aware schedule reduces it to 5.92% (HM G-B-OR). Since a more formal approach to determine balance-aware schedules was out of this work's scope the remaining experiments assume the basic scheme.

Tracking the sub-threads in the ROL is another source of overheads, which is proportional to the total number of sub-threads created. On an average (harmonic), ROL management increases GPRS mechanism overheads to $P_g = 15.49\%$ (G-B-ROL). Dedup, Barnes-Hut, Blackscholes, Reverse Index, and Pbzip2 create a large number of sub-threads (column 7, Table 2) and hence incur higher overheads (maximum of 44.03% for Pbzip2). Pthreads programs, of course, incur no ordering or ROL overheads (not shown).

Result 2. *The checkpointing penalty of conventional CPR (P-CPR) is higher than GPRS due to the use of global barriers.*

The checkpointing penalty is proportional to the amount and frequency of the checkpointed data. To reduce the data size, we used application-level checkpointing. It was applied to both P-CPR and GPRS. We added the necessary checkpointing function at synchronization points. These functions were easily coded in most cases, by replicating the existing computation loops to make deep copies of the data. Blackscholes, Dedup, Histogram, and Reverse Index checkpoint relatively large amounts of data, followed by Barnes-Hut and Canneal. Most sub-threads in Pbzip2 and RE are idempotent, a property we exploited to minimize their checkpointed data. Reverse Index, Dedup and Pbzip2 allocate and deallocate memory in their sub-threads, which add to the GPRS logging overheads. The GPRS also logs the necessary state and information for file I/O operations (almost all programs read from files and three also write to files).

In Barnes-Hut, Blackscholes, Swaptions, Histogram and Word Count, the fork and join points provide convenient spots to checkpoint data for both P-CPR and GPRS. GPRS created sub-threads and checkpointed data in Reverse Index, Canneal and RE at the synchronization points. We emulated P-CPR in Reverse Index, Canneal and RE using barriers. For these eight programs both GPRS and P-CPR used the same checkpointing frequency. The last two bars (P-/-CH and G-B-CH) in Figure 8(a) show the cumulative overhead, including the checkpointing penalty. On average (harmonic), these eight programs incur an overhead of 17.35% in P-CPR and 9.37% in GPRS (not shown).

Dedup and Pbzip2 use a large of number synchronization operations. Emulating P-CPR in these cases, using barriers as often as GPRS checkpoints (at the start of each sub-thread), overwhelmed their execution. Hence, we applied the barrier at the rate of 1/s for Pbzip2 and 5/s for dedup (but did not alter the frequency in GPRS). In Pbzip2, due to uneven work sizes, barriers prove more expensive than GPRS (58.23% versus 50.49%), despite the large disparity between the checkpoint frequencies. In contrast, Dedup's small work sizes and a single long thread kept the CPR overheads relatively lower, 26.31%, as compared with 49.37% for GPRS. Dedup presents an example where GPRS overheads can be higher, due to a large number of small sub-threads.

Overall, on average P-CPR's checkpointing penalty was 21.35% (HM P-/-CH) as compared with the cumulative 15.63% of GPRS

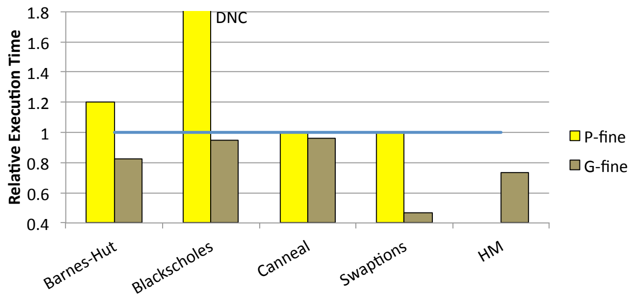


Figure 9. Pthreads and GPRS performance using finer-grained computations. DNC = did not complete.

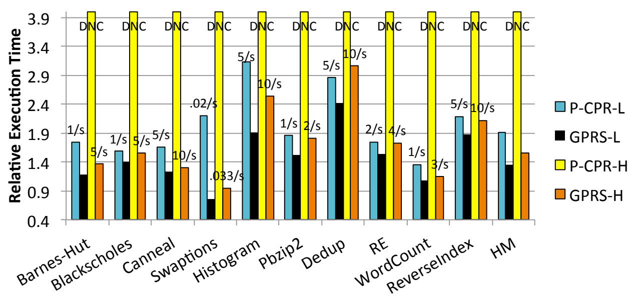


Figure 10. Recovery at different exception rates using conventional CPR and GPRS. DNC = did not complete.

(HM G-B-CH), even though GPRS checkpointed more often for Dedup and Pbzip2. Interestingly, P-CPR's penalty ($P_c = \frac{1}{t} \cdot n \cdot (t_c + t_s)$, t_c = thread coordination cost, t_s = state recording cost) was worse than GPRS despite the ordering and ROL overheads of GPRS. This is because P-CPR uses a full barrier before checkpointing, whereas GPRS can checkpoint when the sub-thread is created. Barriers can severely impact performance, especially for larger number of threads and uneven work sizes. In GPRS, ordering causes a sub-thread to wait for the preceding sub-thread(s), but not for all sub-threads. Hence, the benefits of ordering, which eliminates the checkpointing coordination, outweigh the overheads incurred to implement it, i.e., $t_g \ll t_c$.

Lastly, application level-checkpointing minimizes the state to be saved, i.e., minimizes t_s.

Result 3. *Due to sub-threads the overall GPRS overheads are much lower than the conventional CPR.*

Next we exploit the finer-grained sub-threads in GPRS. The sub-threads in Pbzip2, Dedup, RE, and Reverse Index sub-threads, created at the synchronization points, are already fine-grained. For programs that do not communicate often, we created finer-grained computations by simply launching more threads using the command line argument. Histogram and Word Count already use very small-sized threads. Hence we launched more threads only in Barnes-Hut, Blackscholes, Swaptions, and Canneal. Column 7 in Table 2 shows the total number of sub-threads created in all the programs. Naively creating more threads in Pthreads programs can degrade their performance due to resource contention. Of the four programs, we see this in the case of Barnes-Hut and Blackscholes (Figure 9). While Barnes-Hut degrades by 20%, Blackscholes did not complete (DNC in the figure) in a reasonable amount of time. The load-balancing scheduler (not the same as the round-robin or balance-aware schedule) in GPRS, however, takes advantage of the finer-grained sub-threads and yields better performance. On an average (harmonic) this improves the GPRS execution time by 26.64% (Figure 9) over the baseline Pthreads .

Figure 8(b) shows the execution time using the fine-grained GPRS sub-threads, relative to the Pthreads baseline (column 5, Table 2). We do not compare with fine-grained Pthreads since fine-grained Pthreads is never better than the baseline (Figure 9). Data for Histogram, Dedup, Pbzip2, RE, Word Count, and Reverse Index are the same as the coarse-grain data. Despite creating a large number of sub-threads and hence potentially increasing the management overheads, the overhead trends seen in the coarser-grained experiments still held. Further, the GPRS load-balancing scheduler exploited the finer-grained sub-threads to widen the overall checkpointing overhead gap between P-CPR and GPRS to 21.13% (P-/-CH and G-B-CH HM bars) for the implemented checkpoint frequencies. Note that the GPRS execution time is now the same as baseline Pthreads, despite the overheads.

Handling exceptions using conventional CPR and GPRS.

Result 4. *GPRS substantially outperforms the conventional CPR (P-CPR) and proves just as effective even when P-CPR fails.*

We now examine the recovery capabilities of GPRS and compare them with P-CPR.

Canneal uses non-standard APIs to synchronize in its main computations. Hence GPRS cannot be applied without altering the program. However, to emulate P-CPR we had added Pthreads barriers and the checkpointing functions. We used GPRS to apply hybrid recovery to Canneal, performing selective restart in the checkpointing region of the code and applying P-CPR to the main computations. In all other programs GPRS applied selective restart.

Figure 10 shows the execution time relative to the baseline Pthreads (column 5, Table 2), at two different, low and high, exception rates. The exception rates are listed above each pair of P-CPR and GPRS bars. Whether a program completes is influenced by the sub-thread sizes and the exception rate, as expected. Large sub-threads can tolerate relatively smaller rate of exceptions. For example, sub-threads in Swaptions are large (even when fine-grained). If the exceptions prevent the sub-threads from completing, the program fails to complete. Hence we picked the rates of 0.02/s (low) and 0.033/s (high) to study Swaptions. In contrast, sub-threads in Dedup, Histogram, Canneal, and Reverse Index are smaller, and hence can tolerate higher rates of exceptions. We picked exceptions rates of 5/s (low) and 10/s (high) to study them. We used exception rates of 1/s and 5/s for Barnes-Hut and Blackscholes, 1/s and 2/s for Pbzip2, 2/s and 4/s for RE, and 1/s and 3/s for Word Count. Swaptions' GPRS execution time was lower than the Pthreads baseline despite exceptions due to the fine-grain sub-threads (Figure 9).

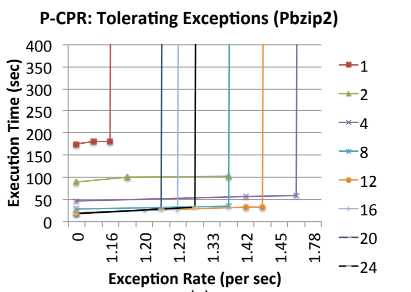
For low enough exception rates, P-CPR completes the execution (P-CPR-L), and so does GPRS (GPRS-L). However, GPRS is 55.44% more efficient, on an average (harmonic), in handling exceptions at the individual chosen rates. Next we increased the exception rate, by a factor ranging from 1.65× to 5× (listed above the GPRS-H bars). In all the cases, P-CPR failed to complete, whereas GPRS completed since it reduces the restart penalty by discarding only affected work. Moreover, GPRS incurred an average additional cost of only 20.70%. The overall GPRS benefits accrue from the reduced checkpoint and restart penalties.

Scalability.

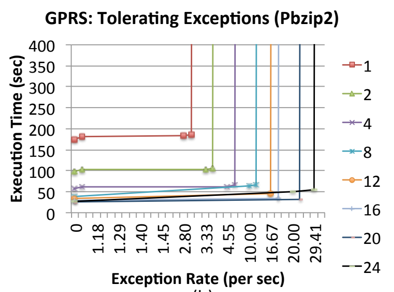
Result 5. *GPRS scales proportionately with the system size whereas conventional CPR (P-CPR) does not.*

As the exception rate grows, the amount of work discarded and hence the restart penalty grows. Beyond a certain rate, for a given computation size, the program will not complete since the same computations keep getting discarded, as seen for high exception rates in Figure 10. We stressed the two approaches to test this limit.

We show the results for Pbzip2, representative of a more generic case, here. Figures 11(a) and 11(b) plot the exception rates (X axis)



(a)



(b)

Bzip2 Tipping Rate

# Contexts	P-CPR Exceptions/s	GPRS Exceptions/s
24	1.32	31.25
20	1.28	22.72
16	1.3	17.86
12	1.43	17.24
8	1.41	11.36
4	1.76	5.62
2	1.41	4.03
1	1.17	1.92

(c)

Figure 11. Exception tolerance of conventional CPR (P-CPR) and GPRS at different exception rates, from 1 to 24 contexts. (a) P-CPR applied to Pbzip2. (b) GPRS applied to Pbzip2. (c) Pbzip2 tipping rate.

for P-CPR and GPRS, respectively, and the execution time (Y axis) of Pbzip2, from 1 to 24 contexts. The restart penalty, and hence the execution time increases with the fault rate until the *tipping rate*, also tabulated separately in Figure 11(c), when the program cannot be completed. For P-CPR (Figure 11(c), 2nd column) this point is around a single exception rate for all contexts, ∼1.5/sec (1.17 to 1.76), as expected from our analysis ($e \leq \frac{1}{t_r}$, e = exception rate, t_r = restart delay). The tipping rate variations in P-CPR as the context-count varies arise from the artifacts of the machine's microarchitecture and the experimental setup.

In contrast to P-CPR, for GPRS the tipping rate scales with the number of contexts (Figure 11(c), 3rd column), from 1.92 to 31.25 exceptions/sec, going from 1 to 24 contexts. This scaling arises from selective restart, as we analyzed in §2 ($e \leq \frac{n}{t_r}$), thus validating its efficacy. Also note that for $n = 1$, both P-CPR and GPRS have the same tipping rate, (∼1.5/sec), as also predicted by the analysis. Similar trends hold for other programs, albeit at different tipping rates and scalability.

In summary, the above results, based on a real machine, demonstrate that global precise-restart helps to reduce exception handling overheads. The results also show selective restart's tolerance to high exception rates where the conventional method fails.

5. Related Work

We achieve globally precise-restartable execution of parallel programs and use it to handle global discretionary exceptions, whereas related work, in general, focuses on a subset of our work's objectives and is not precise-restartable. Table 1 summarizes the related work. Here we discuss the work more closely related to ours.

Fault tolerance based on hardware and software checkpoint-and-recovery (CPR) techniques is a much studied subject. CPR has also been used to manage resources in cloud servers [46]. Most proposals, if not all, take the conventional nondeterministic view of a parallel program, unlike our ordered view. Different proposals trade off design complexity with various overheads. Software proposals [10, 11, 15, 30, 39] preclude the need for special hardware. They use periodic program-wide barriers to perform system-level [27], or application-level, or hybrid checkpoints [30]. These schemes, however, may not be scalable, as we analyzed in section 2. They do not explore handling exceptions in system calls or in their own operations. Our scheme is also software-based and uses application-level checkpointing, but scales and has a wider scope of recovery.

Hardware proposals [3, 34, 37, 43] add considerable system complexity, but can handle exceptions at lower overheads than software system like ours. Some of them also reduce the checkpoint and restart penalties [3, 37] but not to the extent of selective restart. A hardware thread-level speculation-based proposal provides checkpoint-and-rollback mechanism to debug programs [36]. Its notion of partially ordered transactions is similar to our notion of ordered sub-threads. However, its checkpoint size is constrained by the cache capacity. Its application to handle global exceptions was not explored and its suitability for the purpose is unclear.

Several deterministic execution proposals for conventional parallel programs have been made in recent years. Hardware proposals add considerable complexity, and surprisingly high overheads [13, 14, 24]. Software proposals also incur similar overheads [6–8, 28, 35]. Grace works with fork-join type programs [8]. DTHREADS [28], CoreDet [6], and Kendo [35] work with more generic programs, but do not take the parallelism pattern into account when creating the order. As we discussed in §3.2, certain types of programs may incur very high overheads in these schemes. DTHREADS can automatically handle data race-prone programs. Our model can handle data races with help from the user or a dynamic data race detector. Our balance-aware ordering can handle generic programs more efficiently. None of the deterministic proposals actually explore exception handling.

6. Conclusion and Future Work

Techniques to conserve energy and enhance performance in future systems will give rise to discretionary exceptions. These exceptions will frequently interrupts programs. Yet, for the techniques to be effective the exceptions will have to be tolerated, efficiently and automatically. Global discretionary exceptions, in particular, pose challenges in parallel systems.

To handle global exceptions we proposed *globally precise-restartable* execution of parallel programs, analogous to the precise interruptible execution of sequential programs. We presented a software runtime prototype of the approach to handle global exceptions in suitably-written parallel programs. The runtime makes the program's execution deterministic, and overcomes the performance impact of determinism, to be efficient and scalable. Our experiments on a commodity multiprocessor system showed that the runtime easily handled frequent exceptions whereas the conventional method failed. The results quantitatively confirmed our qualitative analysis of the model's performance.

We are currently exploring ways to apply the model to more generic, specifically data race-prone parallel programs, and deterministic/nondeterministic schemes to make it more efficient.

Global precise-restartability can be applied to a wide range of applications, e.g., computer forensics, fault tolerance, debugging, etc. Precise interrupts, once introduced, enabled new capabilities in microprocessors, enhancing their utility. We believe that global precise-restartability can bring similar benefits to parallel systems.

Acknowledgments

This material is based upon work supported, in part, by the National Science Foundation under Grant CCF-0963737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation.

References

[1] "Activities. Android Developr's Guide," http://developer.android.com/guide/components/activities.html.

[2] "Amazon EC2 spot instances," http://aws.amazon.com/ec2/spot- instances/.

[3] R. Agarwal, P. Garg, and J. Torrellas, "Rebound: Scalable checkpointing for coherent shared memory," ISCA, 2011, pp. 153–164.

[4] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: Findings and implications," SIGMETRICS, 2009, pp. 37–48.

[5] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," PLDI, 2010, pp. 198–209.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," ASPLOS, 2010, pp. 53–64.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dOS," OSDI, 2010, pp. 1–16.

[8] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++," OOPSLA, 2009, pp. 81–96.

[9] F. Blagojevic, C. Iancu, K. Yelick, M. Curtis-Maury, D. S. Nikolopoulos, and B. Rose, "Scheduling dynamic parallelism on accelerators," CCF, 2009, pp. 161–170.

[10] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," IPDPS, 2006, pp. 8–.

[11] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," ASPLOS, 2004, pp. 235–247.

[12] N. P. Carter, et al., "Runnemede: An architecture for ubiquitous high-performance computing," HPCA, 2013, pp. 198–209.

[13] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared memory multiprocessing," ASPLOS, 2009, pp. 85–96.

[14] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, "RCDC: A relaxed consistency deterministic computer," ASPLOS, 2011, pp. 67–78.

[15] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Future Technologies Group, White paper, 2003.

[16] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," ASPLOS, 2012, pp. 301–312.

[18] ——, "Neural acceleration for general-purpose approximate programs," MICRO, 2012, pp. 449–460.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," PLDI, 1998, pp. 212–223.

[20] J. Gilchrist, "Parallel data compression with bzip2," http://compression.ca/pbzip2/.

[21] M. Gupta and E. Schonberg, "Static analysis to reduce synchronization costs in data-parallel programs," POPL, 1996, pp. 322–332.

[22] M. S. Gupta, J. A. Rivers, P. Bose, G.-Y. Wei, and D. Brooks, "Tribeca: Design for PVT variations with local recovery and fine-grained adaptation," MICRO, 2009, pp. 435–446.

[23] M. Gupta, K. Rangan, M. Smith, G.-Y. Wei, and D. Brooks, "DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors," HPCA, Feb 2008, pp. 381–392.

[24] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood, "Calvin: Deterministic or not? Free will to choose," HPCA, 2011, pp. 333–334.

[25] "Semiconductor Industry Association (SIA), Design, International Roadmap for Semiconductors, 2011 edition." http://public.itrs.net.

[26] X. Li and D. Yeung, "Exploiting application-level correctness for low-cost fault tolerance." *J. Instruction-Level Parallelism*, vol. 10, 2008.

[27] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," University of Wisconsin, Madison, Technical Report CS-TR-1997-1346, Apr. 1997.

[28] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: efficient deterministic multithreading," SOSP, 2011, pp. 327–336.

[29] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: Models, characterization, and classification," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 703–713, Jul. 1999.

[30] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghil, "Optimizing checkpoint sizes in the C3 system," IPDPS, 2005, pp. 226.1–.

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

[32] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications," DAC, 2012, pp. 1137–1142.

[33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[34] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers," HPCA, Feb 2006, pp. 200–211.

[35] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," ASPLOS, 2009, pp. 97–108.

[36] M. Prvulovic and J. Torrellas, "ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes," ISCA, 2003, pp. 110–121.

[37] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," ISCA, 2002, pp. 111–122.

[38] G. Ramalingam and K. Vaswani, "Fault tolerance via idempotence," POPL, 2013, pp. 249–262.

[39] M. Rieker, J. Ansel, and G. Cooperman, "Transparent user-level checkpointing for the native posix thread library for linux," *The Int. Conf. on Parallel and Distrib. Process. Techn. and Appl.*, Jun 2006.

[40] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," PLDI, 2011, pp. 164–174.

[41] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon, "Innovations in internetworking," C. Partridge, Ed., ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[42] D. J. Sorin, "Fault tolerant computer architecture," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–104, 2009.

[43] 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," ISCA, 2002, pp. 123–134.

[44] B. P. Wood, L. Ceze, and D. Grossman, "Low-level detection of language-level data races with LARD," ASPLOS, 2014, pp. 671–686.

[45] G. Yan, X. Liang, Y. Han, and X. Li, "Leveraging the core-level complementary effects of PVT variations to reduce timing emergencies in multi-core processors," ISCA, 2010, pp. 485–496.

[46] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud," CLOUD, July 2010, pp. 236–243.