Precise-Restartable Execution of Parallel Programs

Gagan Gupta and Gurindar S. Sohi

Abstract

Precise interruptibility enabled a broad range of system capabilities and microarchitectural techniques in instruction-level parallel processors. Arguably, it was key to the success of computers. Multiprocessors lack a similarly-capable feature. As parallelism evolves from instructions to threads or tasks, we envision a similar feature in future multiprocessors.

We define precise restartability of parallel programs, analogous to precise interruptibility. We present a model to achieve instruction-level precise restart of parallel programs executing on multiple processors. The model permits trade-offs between overheads, complexity, and performance based on the expected operating conditions. We present a prototype of the model. Its application to fault-recovery is evaluated on a stock multiprocessor system. Analytical and experimental results showed that the prototype significantly outperformed the traditional approaches, by a factor as high as the number of processors in the system.

1. Introduction

Precise interruptibility (PI), an established feature in processors, is used to handle exceptions (any event that requires a response different from the current flow of program execution). PI support enabled a broad range of capabilities in computers, contributing to the rise of modern computing. PI helps simplify the design and use of computers while permitting performance-enhancing techniques. We believe that multiprocessors, which do not support system-wide PI, can likewise benefit from a similar feature. We present a way to introduce system-wide PI in multiprocessors, and explore its benefits.

Computers did not support PI at first. Systems in the 1960s, e.g., the IBM 360/91, exploited ILP, but lacked PI. Implementing PI was deemed too complex, detrimental to performance, and unimportant since exceptions were thought to be rare. PI’s absence, however, made system debugging onerous, limiting IBM 360/91’s success. PI was subsequently added to the IBM 370 to support virtual memory, and went on to become crucial to other usability-enhancing system applications, e.g., test and debug, detect and recover from faults, manage resources, analyze security breaches, etc.

Processors, too, followed a similar trend. Processor proposals in the early 1980s supported restartability, but not PI [16]. Restartability provided some, but not all benefits of PI. PI techniques were proposed soon thereafter [28]. But, even into the 90s, some processors, e.g., the DEC Alpha 21064, did not precisely handle exceptions that were deemed rare, to avoid the implementation complexity. At about the same time, designers used PI mechanisms to enable innovative microarchitectural techniques, e.g., speculation, as exceptions became frequent, e.g., branch mispredictions. Consequently, ILP techniques evolved from pipelined single-issue to in-order superscalar to dynamic OOO superscalar to thread-level speculation, as hardware exploited more and more parallelism from sequential programs. Over time, PI’s collective benefits outweighed its complexity and overheads, and PI became a de facto feature.

The performance gains from the evolution of ILP techniques have now stalled. Designers have turned to exploiting user-created task-level parallelism (TLP) from parallel programs. In these approaches, the user creates parallelism between tasks, or threads, which may execute concurrently across multiple processors. But these approaches lack system-wide PI, although the system’s individual processors may support PI. Hence, a local exception, one that impacts an individual task, e.g., a page fault, can be handled precisely. A global exception, one that may impact multiple tasks, e.g., a hardware fault, cannot be. Given the ubiquity of multiprocessors and the expected increase in global exceptions, we believe that handling global exceptions efficiently will be important in the future.

Global exceptions, e.g., soft or transient errors, once rare, are now becoming common due to technology scaling and changing use of computer systems [27]. Their occurrences are likely to grow due to aggressive resource management [11], near-threshold voltages [9], process variation, and thermal and voltage emergencies, as has also been noted by others [14, 34, 36]. Researchers are proposing that “failures” due to these factors be treated as exceptions. They propose to process the failures by either treating these events as resource management issues [25, 34] or as intermittent hardware failures [14, 15], or a combination thereof [36]. These solutions implicitly assume PI to address such events in sequential programs. Similar solutions, however, may not be as easily extendible to handle global exceptions in parallel programs.

Present approaches to handle global exceptions in parallel programs make the parallel execution restartable. We note that these approaches are analogous to the restartable techniques of the early 1980s in processors. They enable some capabilities, but cannot match PI’s benefits. Moreover, they incur high overheads. Our qualitative analysis showed that they are too inefficient to manage frequent exceptions, and may even be inadequate in the future. This prompts us to ask: How might we achieve system-wide PI in parallel systems? What might it enable? What would be its complexity and overheads?

We answer the above questions in this paper. We start by analyzing the current global exception handling approaches to make a case for precise-restartable execution of parallel programs, analogous to precise-interruptible execution of se-
quential programs (§2). Challenges to achieving such a feature are analyzed next (§3). We propose a model that introduces an implied order in a parallel program’s execution to provide instruction-level precise restartability (PR) (§4). The model permits parallel execution, but manages the program’s operations and their state to achieve PR when exceptions occur. It also provides the flexibility needed to tradeoff different overheads, performance, and complexity, driven by the frequency and types of exceptions. Qualitative comparison with the current approaches suggests that the model eliminates overheads and handles exceptions more efficiently. We also explore different applications of the model.

We developed a software prototype of the model (§5). Although the model has wide applicability, we applied the prototype to recover from transient hardware faults. The prototype’s evaluation on a real multicore machine showed that compared with the current approach it tolerated more faults (§6).

2. A Case for Precise Restartability

The implicit order in sequential programs helps simplify exception handling in ILP processors. PI makes a sequential program’s parallel execution appear to preserve the program’s implicit order. This helps create a snapshot of the precise architectural state (PAS) during the execution, or across multiple executions on the same or different systems. The PAS reflects the program’s execution only up to the desired instruction. The PAS and the process to create it exhibit four key properties. (1) The PAS is restartable, i.e., the program can resume as if it were not interrupted. (2) The PAS is predictable, i.e., the user can predict the contents of the PAS. (3) The creation of the PAS is repeatable, i.e., the same PAS is generated for the same inputs in the different executions of the program. (4) The PAS can be created at a desired granularity, as fine as instruction boundaries, and by extension, as coarse as, say, procedure boundaries. The PAS can be used to pause the execution for later resumption, or to analyze the program’s behavior. The state’s preciseness permits quick resumption, from the same point where the program paused, and simplifies the analysis. It is the PAS and the four related properties that ultimately enable PI’s broad range of well-understood applications.

Current Approaches to Handling Global Exceptions

The nondeterminism of conventional parallel programs complicates global exception handling in multiprocessors [10]. Such programs do not define a total order on their instructions. Hence, when an exception occurs, the unique point of exception may be determined within a task, but not within the program. This makes creating a PAS difficult, perhaps even moot. Presently, two main approaches are used to handle global exceptions: checkpoint-and-rollback/recovery (CPR) and repeatable execution. While the former is restartable at a coarse granularity, the latter is repeatable at a fine granularity. They obtain these properties by managing the nondeterminism and inter-context communication in the program, incurring periodic, and post-exception overheads. But due to lack of a total order in the program, neither achieves predictability. Hence, although they incur the overheads, they fail to match PI’s benefits. Moreover, as systems become more exception-prone, the overheads will diminish their effectiveness.

CPR. CPR makes the execution restartable by periodically checkpointing the program’s consistent architectural state (AS). The AS reflects a legal execution schedule of the program’s instructions up to the checkpoint [10]. This permits the program’s resumption, but only from the last checkpoint. It is adequate when the AS from a time prior to an exception can be used to resume the program, e.g., to recover from hardware faults, or to migrate programs in dynamic systems.

Coordinated and uncoordinated checkpointing are the two main CPR schemes [10], implemented in hardware and/or software [1, 7, 10, 32]. Different proposals trade off system complexity with the various overheads. Software CPR schemes may be employed within [8, 20] or without [19] the user programs. They may use system-level checkpointing [19], application-level checkpointing [20], or hybrid schemes [7].

CPR schemes incur a checkpoint penalty, $P_c$, every time they checkpoint, and a restart penalty, $P_r$, when they restart from an exception. In coordinated CPR, all contexts that communicated since the last checkpoint coordinate to take a snapshot of the AS. The checkpoint penalty is proportional to the time each context spends coordinating ($t_c$) and checkpointing ($t_s$). Further, contexts may idle until other contexts have checkpointed, before resuming. This may cause a system-wide performance loss ($\Delta_r$). Ignoring actual mechanisms and assuming contexts can checkpoint concurrently, if a fraction $f$ of $n$ contexts communicate in a checkpoint interval, $P_c = f \cdot n \cdot t_c + t_s + \Delta_r$. The total checkpoint penalty is proportional to the checkpoint frequency.

To eliminate the inter-context coordination, uncoordinated CPR schemes take uncoordinated snapshots, reducing their checkpoint penalty to, $P_c = t_s$. Creating the AS from uncoordinated snapshots, however, requires offline analysis. This shifts the coordination overheads to the time of restart, and only when needed, but at the risk of cascading rollbacks [10].

CPR schemes incur the restart penalty when they restart, since all useful work after the last checkpoint is lost. We believe that as exceptions become frequent, this penalty will become critical. Assuming a rate of $e$ exceptions/sec and a checkpoint interval of $t$ sec, the restart penalty (loss of parallelism) $P_r = n \cdot e \cdot t$. 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 $n \cdot e \cdot t \leq n$, i.e., $e \leq 1/t$.

Repeatable execution proposals can reduce the restart penalty to, $P_r = n \cdot e \cdot t$, by limiting the restart only to the contexts ($n_c$) that communicated in a checkpoint interval, where $1 \leq n_c \leq n$ (: $e \leq n_c/t$ for a program to complete) [24]. PR helps reduce this penalty further (§4), and provides other benefits over repeatable systems, as seen next.
Repeateable Execution. Creating the AS using CPR’s coarse-grain checkpointing may not be repeatable. Hence, recent proposals make the nondeterministic execution repeatable at a finer granularity by enforcing a dynamically explicit order between executing contexts, either always or when they operate on shared memory, which otherwise may lead to divergence. This process may be applied in addition to periodic checkpointing. Repeateability is needed to create identity between architectural states of multiple executions of a program, e.g., to test and debug, to analyze for malicious attacks, to detect hardware faults, for forward error recovery, etc.

Both hardware and/or software repeatability techniques [4, 18, 24, 33, 35] have been proposed. While some execute threads in a predetermined manner, others log events during the execution so that they can be replayed in the same order during the re-execution. Other fork-join models execute threads in private spaces until they can be merged, in a sequential order [5].

Although repeatable, the executions may not be precise restartable, may be different from the programmer’s intuition, and may in fact change unintuitively with the changes in inputs or the code. These schemes can also require the replicas to execute concurrently, or on homologous systems, to maintain the identity between the replicas. Enforcing repeatability increases the system complexity. The overhead of this process, the ordering penalty, \( P_o \), in a given checkpoint interval is proportional to the per-context ordering cost \( (t_o) \) and the frequency of order enforcement \( (m) \), and any performance lost when contexts wait for their turn to proceed \( (\Delta_o) \). Therefore, for \( n \) contexts, \( P_o = m \cdot n \cdot t_o + \Delta_o \). This penalty can be very high, and range from 1×-10× in practice [4].

Summary. Unlike PI, the present global exception handling methods provide a subset of the four properties. None is adequate to serve all applications by itself, due to the overheads or the missing properties.

Precise Restartability to Handle Global Exceptions

Therefore, to seamlessly and efficiently handle global exceptions, we argue for precise restartability (PR), with the aim to create the PAS and obtain the four properties during a parallel program’s execution. To achieve PR, we introduce a implicit order between tasks comprising the program. A task is a multi-instruction unit of computation that is scheduled for execution on a processor. We define the execution to be precise restartable if, upon exception, (i) the architectural state reflects the program order execution of the program up to the excepted instruction, but not beyond, and (ii) the execution can resume as if it were not interrupted. While PI was introduced going from ordered sequential programs to their parallel execution, we start with a parallel execution and add order to achieve PR. Assigning an order to tasks may appear antithetical to parallelism, but as we show, order can be overcome without compromising performance, and exploited to handle global exceptions efficiently.

PR offers multiple advantages over present methods. Like PI, PR presents a unified solution to handle different types of exceptions. By definition, like PI, PR is naturally restartable, repeatable, predictable, and applicable at a desired granularity. Further, unlike the present repeatable methods, PR is repeatable whether the replicas execute concurrently or separately, on the same or dissimilar systems. PR’s inherent properties help to reduce the checkpoint, restart and ordering penalties, as we show (§4).

Here we note the evolution of exception handling in processors. Restartable exception handling, which incurred relatively restart penalty, was justified since exceptions were rare [16,17]. Introduction of PI enabled new capabilities. It also lowered the restart penalty, prompting designers to contemplate new techniques, e.g., speculation. Any resulting increase in exceptions, e.g., mispeculations, became tolerable due to the low restart penalty, yielding net gains, e.g., performance. We show the multiple ways PR affords to lower the restart penalty (§4), and present an example of its benefit to multiprocessors (§5).

3. Challenges to Precise Restartability

To study the challenges in achieving PR, we study how systems handle exceptions. To handle exceptions, systems typically manage the program’s execution. The complexity and efficiency of managing the execution in multiprocessors are affected by the types of exceptions handled, the program’s execution schedule, operations performed during the execution, and the system’s scale. PR requirements add further constraints. We describe these aspects and the challenges they pose to a PR global exception handler (GEH).

Exceptions. Exceptions can be of different types and require different responses. For immediate exceptions, those that halt a task immediately, e.g., debug breakpoints or OS interrupts to manage resources, the exact point of exception within the task is known. In such cases, the program may restart from the same point where it halted. In contrast, for delayed exceptions, ones that may not immediately halt the execution, e.g., hardware faults, the exact point of exception may be indeterminate. Hardware faults may be detected with a latency, and may affect multiple tasks in the interim. Other faults, e.g., memory errors, cannot be attributed to any task in the program. In such cases, the program may need to be restarted from an error-free point preceding the exception.

Whereas some exceptions may require the entire program to halt, others may not. A debug exception may require that the program pause while the user inspects its state. Similarly, an exception to manage resources, e.g., to migrate a program, will require the program to pause. For a transient hardware fault, in contrast, it may suffice to only recover the affected task(s), without halting the entire program. Thus, a GEH will need to handle the different types of exceptions and restarts.

Execution Schedule. In addition to an exception’s characteristics, a GEH will need to account for the program’s execution schedule when creating the PAS. The nondeterministic sched-
will need to overcome the issues due to the parallel execution.

Operations. Next, a GEH will need to handle exceptions any time during the program’s execution. A program’s execution comprises program and non-program operations. Program operations include user tasks, non-user operations, e.g., system calls, and I/O operations. Program operations may also process transient events, e.g., interrupts. Non-program operations, implemented in hardware and/or software, e.g., a run-time system, as we shall see, are needed to parallelize the execution and handle the exceptions at the scale of multiprocessors. An exception can affect any of these operations. Each type of operation poses a unique challenge to the GEH, as follows.

Non-program Operations. Although the non-program operations become integral to the execution, they are invisible to the user. But, exceptions during them, e.g., hardware faults, will impact the user-visible state and ultimately manifest as exceptions in user tasks. The GEH will need to seamlessly handle exceptions in such operations.

Forward Progress. The GEH itself will consume memory and compute resources for its operations. It will be imperative that this does not impede the program’s progress.

Repeatability. Another goal of the GEH is to resume the program as if it were not interrupted. Doing so is simpler for immediate exceptions, since the execution is resumed from the same point. But, after a delayed exception, the program restarts from a point preceding the exception, and may re-execute some operations. To ensure correctness, results of these operations would need to be undone before the restart. Not all operations, however, can be undone, e.g., I/O operations to print text. Further, re-executing some operations, e.g., the random() system call, may produce a different result. Moreover, transient events may not recur when the program restarts. Thus, re-executing instructions can alter the program’s execution from the original. Ideally, the GEH would ensure correctness and repeatability in all cases.

Overheads. Finally, managing the execution when the program and its state are dispersed across the system can be complex. It will be desirable to minimize the complexity and the concomitant overheads. Further, adaptability to operating conditions, e.g., exception frequency, will be desirable to tune the GEH’s implementation.

Summary. Thus, the challenge to PR is to handle the different types of exceptions during the different operations of an execution, and account for the operations’ execution schedule, all while minimizing the complexity and overheads.

4. A Precise-Restartable Execution Model

To meet the challenges in making a program’s execution precisely restartable, we base our model on implicitly ordered parallel programs. As we show, an order between tasks and their constituent instructions simplifies each aspect of managing the execution, outlined above, to achieve PR. Historically, parallel programs have been nondeterministic, but recent proposals have argued for ordered programs to exploit parallelism [2,12,13,23], achieving comparable speedups. They define an implicit sequential order for a program’s task, but execute tasks concurrently when possible. Other proposals, based on the fork-join model, execute a parallel program with an appearance of sequential execution [5]. One may also envision assigning an order to tasks in data-race-free task-based models [6], although presently they do not, for the purpose of PR. Interestingly, our results bolster the case proponents make for ordered parallel programs.

Given a total order, each dynamic task and dynamic instruction occupies a unique point in the program’s execution. Hence, when an instruction exception, the exception can be associated with a single point in the program and the restart point can be precisely defined. Given the point of exception, the PAS can now be created, PR’s key requirement.

Execution Schedule. Mechanisms to create the PAS and restart the program will be defined by the system’s execution model, specifically, on the program tasks’ execution schedule. Proposed execution models exploit varying degrees of parallelism. Some exploit limited parallelism [2,5]. Others employ a dataflow schedule [12,13] to overcome the artificial ordering constraint, and maximize parallelism. They permit tasks to execute and modify the architectural state out-of-order (OOO). The OOO models are more challenging for PR due to their complexity. Hence we assume OOO execution to develop the PR model. Nevertheless, the principles will hold in a system that supports ordered semantics. Although the OOO models pose challenges to it, the PR model can exploit the dataflow schedule to enable new capabilities, as we show.

Dataflow Execution. To develop the PR model, we first briefly describe an ordered parallel program’s dataflow execution as proposed by others [12,13]. Their proposals are analogous to the execution in OOO superscalar processors. They view the program as a dynamic sequence of ordered tasks which are candidates for concurrent execution. Tasks are analogous to instructions and the objects they manipulate are analogous to registers or memory locations. Consider an example program’s pseudocode in Figure 1(a). It invokes a task F from a loop. Figure 1(b) shows the dynamic invocations of F as per the program order, and the objects each invocation writes and reads. The OOO models compute the object identities at run time to establish the dependences between the invocations, as shown in Figure 1(c). For example, task F3, which modifies object A, has WAR dependence (solid arrows) on task F1, which reads A. Likewise, F4 has WAW dependence on F1 (dotted arrows). F4 has RAW dependence on F2 (dashed arrows). The OOO models parallelize the independent tasks, thus overcoming the artificial order. Dependent tasks
are serialized. Figure 1(d) shows a schedule they may achieve on a three processors, e.g., independent tasks, F1, F2 and F6, execute concurrently, while F2, F4 and F5 are serialized, in that order. By so honoring the dependences, the execution preserves the program order and is race free. §5 summarizes the mechanics of such an execution [12, 13].

Precise-restartable Dataflow Execution. Although the program is ordered, the above OOO execution can lead to an imprecise architectural state when exceptions occur. For example, in Figure 1(d) F1, F2 and F6 execute in epoch t1. F1 completes in t1, F6 in t2 and F2 in t3. F4 can begin only in t4, once F1 and F2 complete. Thus, F6 completes out of the program order and can leave the architectural state incongruous with the ordered execution (Figure 1(b)) if F4 excepts.

To achieve PR, we exploit the ordered, dataflow, race-free execution, leveraging the resulting properties, but overcome the issue OOO execution introduces. We achieve PR in two steps: first, at the task boundary, and then within the task, at the instruction boundary. Since a task executes as a unit on a processor, we rely on PI-capable processors to provide PR within a task. We are then left to achieve PR at task boundaries.

Here we extend the analogy with OOO superscalar processors. They execute instructions out-of-order, but provide PI by managing in-flight instructions until they retire. They track the order of the in-flight instructions, e.g., by using a reorder Buffer (ROB), and maintain the state they modify separately from the architectural state, using the ROB or a history buffer or a future file [28], or their variants [29]. Upon exception, the instruction order and the separately kept state are used to make the architectural state consistent with the execution up to the desired instruction. We formulate the PR model on a similar philosophy, but adapt it to reflect the scale of multiprocessors.

We augment the dataflow execution model to track tasks and the state they may modify. We introduce the notion of “retiring” tasks. A FIFO Reorder List (ROL), analogous to the ROB, is used to track tasks. At run time, tasks are logged at the ROL’s tail, in the program order, and retired when they complete and reach the ROL-head. Figure 1(e) shows the ROL’s use when the program in Figure 1(a) executes as per the schedule in Figure 1(d). (Ignore the instructions needed to effect the loop, without loss of generality.) At t1, all tasks, F1 to F6, are logged in the ROL since they have all been processed. In t2, F1’s entry has been retired. At t3, although F6 has completed, its entry is held in the ROL, and removed only after t6, once all preceding tasks have retired.

To manage the memory state, we use what is logically a history buffer. Recall that the granularity of a datum is an object. For each task, we identify the objects it modifies, i.e., its mod set. Since the objects may be unknown statically, we establish their identity dynamically at run time. Before a task starts, we clone (i.e., checkpoint) the objects in its mod set. Since the dataflow execution is race free, a clone can be created without interference from other concurrent tasks. The mod set clones collectively form a distributed history buffer. Further, we clone the PC and the registers of the context just prior to invoking the task. When a task is retired from the ROL, the clones are discarded (and their memory recycled).

The clones may be created using copy-on-write or other software/hardware-based schemes. Alternatively, we can use a software/hardware-based future file to hold the modified state until it can be committed to the memory when tasks retire (we do not explore the different cloning options in the paper).

Analogous to OOO superscalar processors, when a task excepts, say, e.g., F4 excepts in epoch t4 (Figure 1(e)), it is halted and the status is recorded in its ROL entry. Other tasks (F3 and F5) continue to execute (epochs t5 and t6). An exception in the retiring task causes the program’s execution to pause. The architectural state modified by younger tasks

---

Figure 1: (a) Example pseudocode that invokes function F. F: (wr_set) (rd_set) write (reads) objects in the set wr_set (rd_set). (b) Dynamic invocations of the function F, in the program order, and the objects they write and read. (c) Dataflow graph of the dynamic function stream. (d) Example execution schedule of the function stream on three processors (P3). (e) Reorder list entries, completed tasks (CT), and retired tasks (RT) at different time epochs.
(F5 and F6) is restored using the clones, in the reverse ROL order, effectively squashing their execution. The architectural state now reflects the program’s ordered execution up to the excepted instruction, i.e., F1, F2 and F3 as completed. F4 as executed up to the excepted instruction, and F5 and F6 as not started. The PC and registers saved in the precise-interruptible processor are sufficient to resume the execution from the instruction. Thus we handle immediate exceptions.

We note that this precise-restartable, OOO task-superscalar execution of a parallel program may be viewed as an evolution of TLS, which is also precise-restartable, but performs in-order task-superscalar execution of a sequential program. Further, our model uses data dependences to avoid the data hazards. TLS admits the hazards, but rectifies them after the fact.

If a processor cannot provide PI, e.g., due to delayed exceptions, the model cannot be precise restartable from the excepting instruction. The model, however, can provide task-boundary PR, by squashing the excepting task. It can roll back even deeper by squashing more tasks. The execution may resume from the oldest squashed task using its cloned PC and registers. If it is impossible to attribute an exception to any in-flight task, the program can always safely restart from the task at the ROL-head. Thus we handle delayed exceptions.

Selective Processing. Like PI, the PR mechanism can enable other capabilities. Some exceptions, e.g., transient hardware faults, may not require that the program pause and a PAS be created. It may suffice to simply re-execute the excepted task and other dependent tasks, without impacting unaffected tasks. The dataflow schedule ensures that concurrently executing tasks are independent. Therefore, we can achieve selective restart by re-executing the affected task(s), on the same or another resource, after its rollback, without squashing any others. Further, the exception can be handled before the excepting task reaches the ROL-head. For example, if task F4 in Figure 1(d) excepts in t4, the following tasks, F5 and F6 need not be squashed; F5 has not yet begun and F6 is independent of F4. F4’s rollback and re-execution may begin immediately in t4, without waiting for older tasks, e.g., F3, to retire.

In another possibility, tasks may be selectively squashed, analogous to squashing instructions on a branch instruction’s mispredicted path in modern processors.

Global Checkpoints. In certain applications, such as execution analysis for malicious attacks, or program migration when handling a “catastrophic” failure which has compromised the system’s integrity, global checkpoints, possibly committed to stable storage, are used. In the PR model, to create a global checkpoint it is sufficient to incrementally checkpoint a task’s mod set when it retires, and the registers and the PC of the following task. Since in our execution subsequent tasks may modify the objects in the task’s mod set by the time it retires, the task’s mod set is cloned immediately after it completes (cloning objects prior to a task’s start may not always be necessary, as is the case here). Only the retiring context needs to checkpoint (and recycle the clone memory), while others may continue to perform work. Given the program order, the task at the ROL-head is always the oldest task and the global checkpoint is always consistent with the ordered execution.

Non-program Operations. Every non-program operation, e.g., to create the dataflow graph (Figure 1(d)), or a PR operation described above, is performed on behalf of one or more user tasks. For the purpose of PR, we use the program order to merge the operation with the corresponding oldest task. For example, when a completing task communicates with a dependent task, the operation is merged with the completing task. An exception in the operation causes the older task, and hence the operation, to re-execute. The actual details will depend on the model’s implementation, which we describe for our prototype (§5).

Forward Progress. PR execution terminates an excepted task and handles the exception when its entry reaches the ROL-head, requiring preceding tasks to complete in the interim. Despite this requirement and possible arbitrary terminations, the model is deadlock free. Since tasks are introduced in the system as per the program order, a given task is always independent of its successors. Hence the execution always makes progress and an excepted task will always reach the ROL-head. The program can safely resume once the exception is processed and the PAS is restored.

PR operations also use additional resources, e.g., memory consumed by clones, until tasks retire. Once again, the program order can be exploited to prevent deadlocks related to resource exhaustion. The resource usage can be regulated by throttling the execution, waiting for older tasks to complete before introducing younger tasks, without impeding the program’s forward progress.

Repeatability. When the restart after a delayed exception rolls back operations, I/O operations can pose a challenge. If the I/O operations are idempotent, e.g., file read and writes [26], they can be simply re-executed without compromising the functionality. For non-idempotent operations, e.g., network I/O, the model permits the users to enclose them in non-PR tasks. Whereas other tasks may execute in dataflow order, such a task strictly executes in the program order, when it reaches the ROL-head, obviating the need for rollback. The output-commit problem is overcome by waiting for the exception-detection latency before committing the data to output [30].

To ensure repeatability of system calls and transient events, the events, their outcome, where applicable, and the associated task ID can be recorded. The events can be played back or their recorded outcomes supplied to the re-executing tasks.

Granularity. Although we have depicted cloning at task boundaries, it may be performed at sub-task boundaries. This would enable a finer grain restart of tasks, which may be desirable for high-frequency exceptions. Alternatively, the cloning interval may be increased to multiple consecutive tasks, effectively reducing the amount of cloning, e.g., an object modified
by multiple tasks in an interval need be cloned only once, more apt for less frequent exceptions. Thus the model’s implementation can be adapted to operating conditions.

**Other Applications.** One may also apply the model to other applications. For example, the OS can use selective restart for more flexible scheduling, and redundancy-based systems can detect faults by comparing the results of completed tasks, identified by their unique order, between replicas.

**Overheads.**

We now assess PR’s overheads and show that they are lower than those of the conventional approaches.

Since the execution is race-free, PR eliminates all inter-context coordination, unlike the CPR schemes. Any communication (non-program operation) is relegated to the task boundaries. Hence a task’s mod set can be cloned independently, and without pausing other tasks. Since the ROL tracks tasks in the program order, the PAS can be created by simply restoring the requisite clones, without any pre- or post-exception coordination. Ordering ensures that PAS creation always succeeds, avoiding cascading rollbacks altogether. Hence PR’s checkpoint penalty is merely the checkpointing duration, $P = t_s$, the same as that of uncoordinated CPR, but without its drawbacks. (In coordinated CPR, $P = f \cdot n \cdot t_c + t_s + \Delta_c$.)

To create global checkpoints, no system-wide barriers or other coordination is needed. Hence, global checkpointing overhead is also comparable to that of the uncoordinated CPR.

Since PR is naturally repeatable, no ordering is enforced when tasks execute. The model does apply the implicit order at task boundaries. Hence its ordering penalty ($n$ contexts, ordering cost $= t_o$), $P_o = n \cdot t_o$ (in repeatable execution, $P_o = m \cdot n \cdot t_o + \Delta_o$).

For immediate exceptions, PR permits older tasks to complete, and discards only tasks younger to the excepted task, analogous to exception handling in modern processors. In contrast, conventional CPR discards all current work. If on an average half of the in-flight tasks are younger to the excepted task, PR’s restart penalty is half of that of the CPR system.

Selective restart provides further advantage. Since only the excepting context need restart, no work from any other context is discarded. This is 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. Hence in an $n$-context system executing tasks $t$ seconds in size, at a rate of $e$ exceptions/sec, the loss of parallelism (restart penalty) is always constant, $P_r = e \cdot t$. For a program to successfully complete in such a system, $e \cdot t \leq n$, i.e., $e \leq n/t$. Thus selective restart is $n$ times more exception-tolerant than a conventional CPR system ($e \leq 1/t$), and $n/n_e \times$ more tolerant than repeatable systems ($e \leq n_e/t$), making it more effective in highly parallel/exception-prone systems. This low restart penalty may permit users to explore new techniques, an example of which we show ($\S 5$) and evaluate ($\S 6$).

PR provides additional choices that can help strike different tradeoffs. Coarse-grain cloning may be used to reduce the total checkpoint penalty, but at a higher restart penalty (due to the deeper rollbacks), tolerable when exceptions are rare, akin to the checkpoint repair in early processor proposals [17]. Alternatively, finer grain cloning may be used to reduce the restart penalty, but at a higher total checkpoint penalty (due to the increased frequency), more apt when exceptions are frequent. This is akin to handling branch mispredictions in modern processors. Note that PR overheads will be lower than those of a comparable present-day system.

**Summary.**

Thus, by exploiting the program order and checkpointing individual tasks, we achieve restartability, precisely from an instruction or a checkpoint-boundary. Ordering helps manage the program’s execution schedule, uniquely identify the exception and restart points, simplify the PAS creation and restart, handle different types of exceptions in the program’s different operations, all while minimizing overheads and ensuring forward progress. Dataflow execution, itself enabled by the program order, and the checkpointing flexibility help trade off the different penalties in an implementation.

5. Prototype Implementation

Before investigating specific hardware and/or software techniques for the PR model, in this paper we study the model’s viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults.

The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults. The PR prototype is in the form of a C++ run-time library. The library provides APIs to develop a target program, effects viability and the requisite mechanisms. For this purpose we built an all-software PR prototype and applied it to backward error recovery [30] from frequent transient hardware faults.
user code is developed using the runtime’s API (as seen below). But it cannot access the state held by third party libraries and the OS. Instead of re-developing the entire system software, we show a few example implementations.

We explore two fault recovery systems in the PR runtime. The first, Base-PR, resembles a conventional CPR system in which all work since the last checkpoint is discarded prior to recovery. The second, Selective-PR, performs selective restart. The prototype details are described next (§5.1), and the two recovery systems thereafter (§5.2). Evaluations follow in §6.

5.1. Prototype Components

Parallel Dataflow Execution

We incorporated the dataflow execution engine of an existing proposal and its API [13] into the PR runtime. To program using the API, the user invokes df_execute call, for parallel execution. Figure 2(a) shows the code for the example in Figure 1. The dataflow task (DF task) F is invoked on line 4. The user also formulates a DF task’s read and write sets, which include the objects possibly shared with other concurrent task(s) (wr_set, rd_set on line 4; ignore line 5 for now).

The program’s dynamic tasks form a sequence of DF tasks and segment tasks (program code between successive DF task invocations). Figure 2(b) shows the program’s dynamic segment and DF tasks, and the dynamic runtime operations.

The execution begins sequentially, but aims to execute DF tasks concurrently with the other executing DF tasks and the following segment task. As the execution unfolds, a df_execute causes the runtime to initiate the prelude phase (line 12), which establishes the DF task’s data dependence on currently executing DF tasks, using the dynamically computed read and write sets. It submits independent DF tasks for execution and shelves dependent DF tasks. In either case, the execution proceeds to the next segment task, repeating the process. After a DF task has executed, the runtime initiates the postlude phase (Figure 2(b), line 17), which submits any shelved DF tasks, that may now be ready, for execution.

The execution engine is implemented using threads, each of which uses a work deque and a task scheduler. Work is submitted for execution by enqueuing it in the work deque. The engine also uses its own thread-local memory allocator.

Managing Program Operations

When the execution is parallelized, the runtime PR operations (ROL processing, cloning, etc.) manage the program’s tasks. The first PR operation, PRpostSeg, invoked at a segment task’s end, manages the segment task’s completion and the following DF task’s start (e.g., line 10 in Figure 2(b)). The second, PRpreDF, invoked before a DF task, sets up the following segment task (e.g., line 11). The third, PRpostDF, manages a DF task’s completion (e.g., line 18).

Managing Program State. The PR operations also manage the program’s state. The state comprises context-private data and global data, e.g., in Figure 2(b) variable i and the processor state (not shown) are private, whereas variables a, y, β, γ, ζ, and η are global. The PR runtime automatically clones the private state by cloning the context’s stack.

To manage the global state, the prototype tracks the global objects that are live (as defined in dataflow static program analysis) at a task’s start and are modified by it, e.g., the DF mod (DM) set, {B, C} for F1 (line 13), and the segment mod (SM) set, {γ}) for the segment task immediately after F6 (line 62). These objects are cloned to rollback and restart a task.

Although computable by a compiler [7], our prototype relies on the user to specify the above data sets via the extended df_execute API (Figure 2(a), line 5), as do others [12, 13, 20, 23]. We also rely on the user to provide constructors (and destructors) to clone, declone and restore the global data, like other object-oriented programming systems. All data are allocated using the runtime’s allocator.

Managing Runtime Operations

In addition to program operations, the PR runtime manages its own operations and state for fault recovery. It logically merges the PR operations with the related tasks, as depicted by the boundaries in Figure 2(b). It treats PRpostSeg as part of the preceding segment task. PRpreDF, prelude, postlude, and PRpostDF are combined with the associated DF task. The PR runtime recovers from faults at these boundaries. Scheduler operations are treated independently, as we see later.

Managing Runtime State. To manage the execution, the PR runtime uses data structures that are either context-private, e.g., the memory allocator structures, or shared, e.g., the work deques. We once again use the program order to manage them.

A runtime data structure is not cloned. To restore it, we either: (i) reset it to a consistent state, e.g., the ROL, or (ii)
“repair” it to its pre-fault state, e.g., the memory allocator, or (iii) reconstruct it to its pre-fault state, e.g., the work deques.

Inspired by the Aries [22] crash-recovery mechanism, used in DBMS, we use a write-ahead-log (WAL) to repair or reconstruct a runtime data structure. Each task is assigned a unique ID, as per its program order. Before operating on a structure, each context independently logs the logical and/or physical PR operation, and the associated task’s ID, to the WAL, maintained on error-free stable storage. To restore the runtime’s precise state, we sequentially replay the PR operations from the log, without actually performing the program tasks.

Third Party/OS Functions. We managed the memory allocator state using the WAL, as described next. OS directory operations were handled similarly. GCC’s file operations were handled by including the file descriptors in the user data sets.

5.2. Base-PR and Selective-PR Fault Recovery Systems

We now describe the fault recovery in the two systems. Contexts log their PR operations in separate WALs; we refer to WALs collectively as the “log”. Each context logs only key information: the runtime operation’s type and its associated task’s ID, a task’s status, and the executing task’s ID. It also logs the memory allocator’s operations and key physical states.

The PR runtime creates an empty ROL at the program’s start. Except for the very first task, it maintains two invariants: (i) before a task starts, all information necessary to restart it has been created, as part of the preceding task, and is fault-free, and (ii) the ROL-head holds fault-free information, if it exists. A task is deemed completed after it finishes without faulting.

Base-PR Operations

Program and Segment Task. The program’s execution begins with the first segment task. The runtime logs the segment task’s ID, creates an entry for it in the ROL, and clones the private state and the SM set. The task is then executed.

As the execution advances, a df_execute signal resets the segment task’s end and the next DF task’s start. The runtime (PRpostSeg) logs the segment task’s completion, if fault free, and if the segment task is at the ROL-head, retires it (described later). It then logs the DF task’s ID, creates its ROL entry, clones its local state, and processes the function, as below.

DF Task. The runtime (PRpreDF) first processes the segment task that follows the DF task, as described above. Next, if the DF task is independent (prelude), the segment task is pushed in the work deque and the DF task is processed. Otherwise, the DF task is shelved and the segment task is executed.

The DF task is executed after its DM set is cloned. When the DF task finishes, any dependent DF tasks that may now be ready are pushed in the work deque (postlude). If the process was fault free, the DF task’s completion is logged. If its entry is at the ROL-head, the DF task is retired (PRpostDF).

Work Deque. As it seeks work, a thread scheduler obtains a task from a deque. The task’s ROL entry has already been created and is processed as described above.

Retiring Tasks. When a completed task retires, the runtime logs the fact, deletes the task’s clones, and dequeues the entry from the ROL. It repeats the process for any successive ROL entries until it reaches an incomplete task, after which it proceeds to seek more work from the work deques.

Fault Recovery. When a fault occurs, the runtime stops the program, aborts all executing tasks, and initiates recovery. It makes the ROL consistent with the retired tasks in the log, and retires tasks that can be. It walks the ROL in the LIFO order, restoring data using their clones. Next, it repairs the memory allocator using the log. Since all tasks are terminated, all runtime data structures (work deques, ROL, logs), except for the ROL-head and its entry in the log, are reset. The program state now reflects sequential execution up to the (only) task in the ROL, from where the program may be resumed. A fault in the very first segment task causes the program to re-execute.

Selective-PR Operations

Selective-PR is similar to Base-PR, but for the additional logging needed for selective restart. The runtime logs the PR operations performed to manage the dataflow schedule, to access the work deques, and each task’s current phase (prelude, execution, postlude).

When a non-work deque operations faults, the runtime pauses the program and initiates recovery. It identifies the affected data structures using the task ID logged by the faulted thread and its dependent tasks. These structures are then restored using the logs. The affected tasks’ data are then restored using the clones. Once the recovery is complete, all paused tasks resume while the affected tasks restart.

When a work deque operation faults, the affected deque is reconstructed by replaying the operations performed on it after the last retired task. Other steps remain the same.

6. Quantitative Evaluation

We used the two PR systems, Base-PR and Selective-PR, to recover from transient hardware faults “injected” during the program’s execution. We evaluated the systems’ overheads and fault tolerance at different fault rates. Details of the experimental setup, the fault model, and the results follow.

The following programs, commonly used to evaluate parallel systems, were developed using the runtime library: barnes, blackscholes, bzip2, histogram, swapsets, and Redundancy Elimination (RE). RE processes packets to compress the network traffic [3]. The library’s APIs were used to parallelize the code, specify the data sets, and provide the cloning-related functions. The programs were executed on a stock, 2-way hyper-threaded, 6-core Intel Xeon E5-2420 (Sandy Bridge; 32KB each, L1 I and D, 256KB private L2 caches per core; 15MB shared L3 cache)-based, two-socket (total of 24 contexts) machine, running Linux kernel 2.6.32. The programs were compiled using gcc 4.6.1, with the -O3 and march=corei7-avx options. All programs were tested using their standard “large” inputs. RE processed 1.2M packets.
Fault Model and System Assumptions
Many fault-detection techniques have been proposed [21, 31], and are beyond the scope of our work. We assume that mechanism(s) in the system detect faults and report the faulting context to the prototype, with a maximum latency of 400,000 cycles (as have others [32]). The PR systems emulate this by launching an additional context, which uses Pthread signals to periodically signal the other contexts and randomly designate one of them as “faulted”. Since not all third party and OS components in the software prototype are fault tolerant, we blocked the fault signals during their execution, effectively delaying the fault signals until it was safe to process them.

We also assumed the availability of stable storage to maintain the logs. Recovery from a corrupt stable-storage, faults that cannot be attributed to a context, and system-level policies, e.g., handling permanent, repeating or non-recoverable failures are beyond the scope of our work.

Our experiments assume a single-fault model. We stress-tested the system under various fault rates (without emphasizing the distribution). We summarize the key results by analyzing the graph bars in Figure 3. The figure shows the overheads incurred by the fault-tolerant programs over their non-fault-tolerant versions (baseline) for the given context count. We studied the impact on overheads under the following conditions: (i) increasing context count, (ii) recovery using Base-PR (closest to the conventional CPR) and Selective-PR (selective restart), and (iii) increasing fault rates. Although Base-PR’s restart penalty is comparable to CPR’s (both discard all work at recovery), its checkpoint penalty is lower than CPR’s, since contexts in it do not coordinate to checkpoint. Each graph title gives the program name and the total number of tasks it executes. The Y-axis plots the overheads in percentage. The X-axis plots the context count and lists the baseline execution time in seconds (wall-clock time, including all file I/O operations) underneath. The discussion is divided into basic (no faults) and fault recovery overheads. Hereafter, BPR refers to Base-PR, and SPR to Selective-PR.

Basic Overheads
The NSB bars show BPR overheads when no faults occur and signals are not blocked. Thus they show the basic PR overheads, which arise from the PR operations (PRO) and checkpointing. The PRO overheads are small (<10%). The checkpoint penalty is program and context count dependent.

Since bzip2, histogram, swaptions, and RE mostly perform idempotent computations, they require minimal cloning. Hence, they expose the PRO overheads. PRO overheads are fairly constant in these four programs, with the maximum of ~10% (RE, 24 contexts). Barneshut and blackscholes perform a large number (150K and 400K) of tasks, and still incur low PRO overheads - their overheads mainly arise from the checkpoint penalty, discussed next.

Barneshut and blackscholes checkpoint large amount of data. The NSB bars at one context expose their checkpoint penalties, ~8% and ~25%, respectively. In general, BPR’s checkpoint penalty is similar to SPR’s, across all contexts, as expected from their similar checkpointing mechanisms. As the contexts increase, a program’s the execution time and the checkpoint penalty decrease, but the penalty can be up to 110% of the execution time (blackscholes, 16 contexts). The harmonic mean of the total basic overheads across all programs and contexts is 25% (not shown).

$BPR(0)$ and $SPR(0)$ show the overheads at 0 faults, but with signals blocked around third-party functions. Hence they incur the signal-blocking overheads, proportional to the frequency of the calls. Barneshut, blackscholes, swaptions and RE perform file operations, incurring large blocking-related overheads, up to ~80% (RE, 24 contexts). $SPR(0)$ incurs the logging overheads for selective restart, over and above the $BPR(0)$ overheads. Data shows it is almost negligible in most cases, but up to 15% (RE, 24 contexts).

Fault Recovery Overheads
The $BPR(e)$ and $SPR(e)$ bars show overheads at e faults/sec. When compared to overheads at 0 faults/sec, they denote the restart penalty. Program characteristics influence the restart penalty. Results show that when BPR incurs a relatively large restart penalty, SPR can reduce it, especially at larger context counts, as predicted by our model ($\S$4).

Barneshut and blackscholes, due to their small-sized tasks, can easily tolerate fault rates of 1/sec and 2/sec in both runtimes. The total number of faults handled are listed on top of the bars (the setup does not achieve the specified fault rate perfectly). In both programs, a total of only a few thousand computations get discarded and hence both BPR and SPR yield similar execution times, in general, across all contexts. The restart penalty can be up to ~35% (blackscholes, 16 contexts). The large number of tasks in blackscholes and their checkpointing (the most among all programs) create an effect of relatively large total checkpoint penalty. But since the task size is small, work lost at recovery is low, and hence the BPR restart penalty is low. Since the checkpoint penalty dominates the restart penalty, SPR is unable to do much better than BPR for a given fault rate and context count. Further, increasing the fault rate from 1/sec to 2/sec has little impact on the total restart penalty due to the small task size.

Bzip2, swaptions, and RE present a different scenario due to their relatively large, small number of total tasks, and little checkpointing. This effectively creates a relatively small total checkpoint penalty, but a larger BPR restart penalty (more work is lost at recovery). While both BPR and SPR start out with similar execution times for one and two contexts, SPR performs consistently better as the fault rate grows, especially at higher context counts (e.g., the rightmost two bars for each context count - lower is better). In the case of BPR the problem is compounded since increased execution time increases exposure to more faults. Note that, in general, for the three programs, while $BPR(e)$ grows, $SPR(e)$ remains relatively constant, close to $SPR(0)$ and $BPR(0)$ for a given context count. These results highlight selective restart’s ability to handle more
exceptions by reducing the restart penalty.

While the other programs comprised fixed-size tasks, histogram creates only as many tasks as contexts and hence its task size grows inversely with the number of contexts. As contexts decrease, the tasks decrease. Hence the total checkpoint penalty decreases while the BPR restart penalty grows. The program fails to complete at fault intervals shorter than the growing task sizes (shown as ∞ in the graphs). We present data for histogram at 5 and 10 faults/sec. SPR fares better at these rates, although for small number of contexts, one and two, the task sizes are too large even for it to complete.

As the fault rate increases, for a given task size, the restart penalty grows. Beyond a certain fault rate BPR fails to complete the program. Figure 4(a) shows an example. It plots the execution time of bzip2 and swaptions at fault rates varying from 0.03/sec to 2/sec, on 20 contexts. The execution time grows until at 0.71/sec BPR fails to complete bzip2, and swaptions at 0.06/sec. SPR, however, maintains a stable execution time for the ranges shown. Similar trend holds for the other programs, and at other context counts.

Next, we stressed the two recovery systems to test their fault

tolerance limits. Figures 4(b) and 4(c) plot the fault rates (X axis) for BPR and SPR, respectively, and the execution time (Y axis) of bzip2 for 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 4(d), when the program cannot be completed. For BPR (Figure 4(d), 2nd column) this point is around a single fault rate for all contexts, ~1/sec (0.67 to 1.12), as expected from our analysis (e ≤ 1/t). The tipping rate shifts to higher fault rates with increase in the number of contexts for SPR (Figure 4(d), 3rd column), 6.8 to 71.43 faults/sec, going from 2 to 24 contexts, also as expected (e ≤ n/t), thus validating selective restart’s efficacy. Also note that for n = 1 context, both SPR and BPR have the same tipping rate, (∼1/sec), as also predicted by the analysis.

In summary, the above results, based on a real machine, demonstrated the PR model’s viability and its reasonable overheads. They showed selective restart’s tolerance to high fault rates where the conventional CPR would fail. As future computers provide more contexts and exceptions increase, either due to hardware failures or new techniques, PR can provide a low-overhead approach to handle exceptions.
The state can be used for a range of purposes, from program to fault tolerance. The model permits tradeoffs between overheads, complexity, and performance.

7. Conclusion
We defined and proposed a model for precise-restartable parallel execution of ordered parallel programs, analogous to precise interrupts. The model creates a restartable, predictable and repeatable precise architectural state, at a desired granularity, to handle exceptions during a program’s execution. The state can be used for a range of purposes, from program debugging to fault tolerance. The model permits tradeoffs between overheads, complexity, and performance.

Experimental results using the model’s prototype on a stock multicore system show its viability and tolerance to substantially higher rates of exceptions than the present approach. Precise interruptibility enabled a host of capabilities in processors, enhancing their utility. We believe that multiprocessors can similarly benefit from precise restartability.

References