Scalable Fault Tolerance in Multiprocessor Systems

Gagan Gupta
Department of Computer Sciences
University of Wisconsin-Madison
Email: gagang@cs.wisc.edu

Gurindar S. Sohi (Advisor)
Department of Computer Sciences
University of Wisconsin-Madison
Email: sohi@cs.wisc.edu

***Abstract*—Evolving trends in design and use of computers are resulting in fault-prone systems which may not execute a program to completion. Checkpoint-and-recovery is commonly used to recover from faults and complete parallel programs. Conventional checkpointing-and-recovery can incur high overheads and may be inadequate in the future as faults become frequent. We propose to execute parallel programs deterministically to tolerate faults at lower overheads and scalably.**

I. Introduction

Today multiprocessors and hence parallel programs have become ubiquitous. Hardware faults, which can corrupt programs, are slated to become more frequent [1]. To conserve energy and scale performance, newly proposed techniques, such as hardware energy management, dynamic resource management, and approximate computing, can frequently cause errors and corrupt the program's state. Here, we term all hardware and software events that can corrupt a program's state, as *faults*. To successfully use these systems and employ the new emerging techniques, the challenge is to efficiently recover from the faults and still complete the programs.

To recover from faults in parallel programs checkpoint-and-recovery (CPR) approaches [2] periodically checkpoints the program's state during its execution. Since conventional parallel programs are characterized by inter-thread communication and nondeterminism, inter-thread coordination is needed to perform each checkpoint. When a fault occurs, the most recent error-free consistent architectural state is constructed from the checkpoint, from where the program is resumed. This effectively rolls back the execution. Uncoordinated checkpointing, an alternative, can eliminate coordination, but can cause cascading rollbacks [2], and is hence not preferred.

The CPR process incurs overheads from two main sources: (i) periodic inter-thread coordination, and (ii) loss of parallelism when all work since the last checkpoint is discarded, whether "good" or "bad". Due to these overheads CPR may not scale to handle frequent faults. Intuitively, if the rate of faults (e) is more than the work performed in a checkpoint interval (t), a program may never complete, i.e., $e \leq \frac{1}{t}$ for the program to complete, irrespective of the system size, although e grows with the system size. Simply increasing the checkpointing frequency, i.e., $\frac{1}{t}$, will increase the inter-thread coordination overheads, and hence is not appealing.

Here we draw inspiration from CPR, database recovery methods [3], and out-of-order (OOO) superscalar processors to propose a low overhead, scalable fault tolerance model.

OOO processors execute instructions concurrently, often speculatively, and are yet precise-interruptible. Precise-interruptibility allows them to handle *exceptions*, e.g., misspeculations, which can be very frequent, efficiently. The key to precise interruptibility is the ordered view of the instructions the processor executes. We apply an analogous approach in multiprocessor systems in which the parallel program's execution is made deterministic to achieve *global precise-restartability*. Global precise-restartability in turn leads to efficient fault tolerance.

The proposed approach is implemented as an application-level recovery system, called the *Globally Precise-Restartable System* (GPRS). GPRS is a C++ run-time library and can be applied to different styles of parallel programs. It can work with conventional multithreaded programs (Pthreads-based presently) as well as statically-ordered parallel programs [4]. GPRS is also uniquely capable of tolerating faults in the user program as well as its own operations. Presently GPRS is operational on shared-memory systems.

II. Proposed System

In this section we briefly describe how GPRS operates to recover from faults. It manages the program's execution, the program's state, and shepherds the execution to completion when faults arise.

A. Fault Tolerance in Multithreaded Programs

GPRS makes a parallel program's execution deterministic to simplify fault recovery. It divides the threads in conventional, data race-free multithreaded programs into sub-threads at communication points therein. A logical order is assigned to the sub-threads, which are then scheduled for execution in that order. This leads to deterministic execution. However, ordered scheduling can severely impact the performance. We propose various ordering schemes, ranging from simple round-robin to more sophisticated balance-aware, to minimize this impact to less than 5% on an average for ten popular parallel benchmarks [5].

The implicit order precludes communication from "younger" sub-threads to "older" sub-threads. hence an affected sub-thread cannot corrupt older sub-thread(s).

The deterministic execution is then made *globally precise-restartable*, analogous to precise-interruptible sequential programs. Briefly, GPRS tracks (i) the order of the program's currently executing sub-threads, (ii) the objects they may modify and (iii) the state of those objects before they are modified (uncoordinated checkpoint). When a sub-thread faults, objects modified by it and by those "younger" to it can be restored to their pre-modified state, causing the program state to reflect

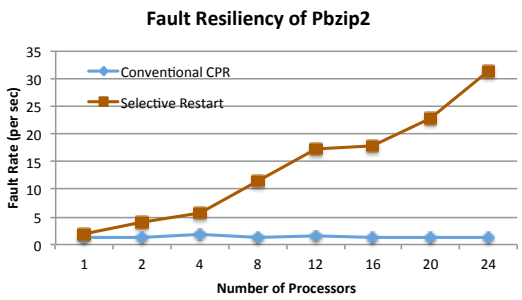

Fig. 1. Fault resiliency of conventional CPR and selective restart for the nondeterministic implementation of the Pbzip2 program running on a 24 processor machine. Selective restart can handle higher rates of faults.

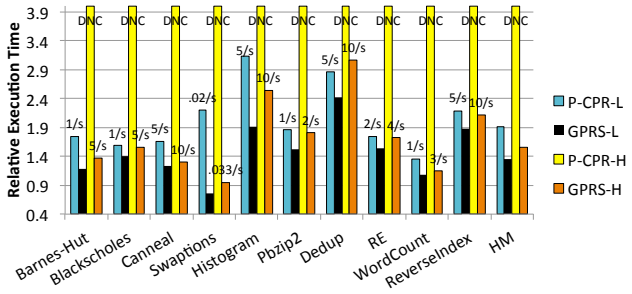

Fig. 2. Relative execution time when recovering at different fault rates (listed above bars) using conventional CPR (P-CPR) and GPRS for different programs. DNC = did not complete; HM = harmonic mean. GPRS completes programs at high fault rates, unlike CPR. GPRS overheads are lower.

precise, ordered execution up to the exception. The program may restart using this state, and is hence *globally precise-restartable*.

The logical order is exploited to selectively re-execute only the faulted sub-thread, without impacting the rest of the program, since only independent sub-threads execute concurrently. Since not all, but only the affected sub-thread may have to be restarted, this enables continuous, online recovery. Further, now for the program to complete, $e \leq \frac{n}{t}$ in an n-context system, which is up to $n\times$ more resilient than conventional CPR (Figure 1). Since GPRS performs uncoordinated checkpoints (without suffering from cascading rollbacks), its overheads are lower than CPR (Figure 2). Thus GPRS performs low overhead, scalable recovery, making it well-suited for the highly fault-prone future systems.

B. Fault Tolerance in Statically-ordered Parallel Programs

Recently researchers have proposed statically-ordered programs for multiprocessor systems. These programs specify an implicit order between the program's tasks, but execute the tasks concurrently whenever possible. GPRS leverages this aspect to simplify fault tolerance in such programs. Since the program already contains the notion of order, no explicit ordering need be enforced, unlike in the multithreaded programs. Global precise-restartability is achieved using the mechanisms described above. GPRS provides scalable fault resilience for this programming model also.

C. Fault-tolerant Runtime System

GPRS being a software system its mechanisms are susceptible to faults. It uses an Aries-like recovery protocol [3], but simplifies the process using the sub-thread order, to recover from faults within the runtime system. Briefly, each operation performed by GPRS can be viewed as performed on behalf of the user program's sub-thread. GPRS logs its operations (on the same stable storage as the checkpoint). Since each sub-thread is ordered, each thread can log the operation concurrently without inter-thread coordination. During recovery, the runtime's state can be reconstructed with the help of the log and the order of the operation. The overheads of this process are reflected in Figures 1 and 2.

For its functioning, GPRS intercepts the programming APIs (of both, Pthreads and the statically-ordered model [4]) and implements its own fault-tolerant versions of the APIs. GPRS also implements its own versions of fault-tolerant memory allocator and system functions, e.g., file I/O, commonly used by programs.

III. Conclusion

Multiple sources of faults, ranging from hardware failures to approximate computing, can prevent a program from completing execution. We propose to use ordered parallel programs or make a conventional parallel program's execution deterministic, and introduce a notion of global precise-restartability. This approach simplifies fault tolerance and reduces the related overheads. It scales with the system size whereas the conventional checkpoint-and-recovery does not.

So far we have tested the proof-of-concept prototype in a relatively small system comprising 24 processors. Our goal is to extend the approach to larger system and explore related mechanisms. Specifically, we plan to combine GPRS principles with proposals like containment domains [6] and transactional memory, to provide efficient recovery on larger systems.

References

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1177/1094342009347767

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[3] 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, pp. 94–162, March 1992. [Online]. Available: http://doi.acm.org/10.1145/128765.128770

[4] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on mulcticore architectures," in *International Symposium on Microarchitecture*, ser. MICRO '11, 2011.

[5] G. Gupta, S. Sridharan, and G. S. Sohi, "Globally precise-restartable execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594306

[6] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *the Proceedings of SC12*, November 2012.