Blocking Application-level Checkpointing for OpenMP Programs

Greg Bronevetsky, Martin Schulz, Peter Szwed, S. Shafat Zaman
Department of Computer Science
Cornell University, Ithaca, NY 14853

I. Introduction

Recent trends in parallel computing are moving us towards systems where reliability is becoming a significant concern. In the hardware arena we are moving from monolithic, big-iron machines to systems that are typically large collections of commodity parts. Furthermore, the applications being run on these machines are becoming longer lived, resulting in the mean running time of parallel applications exceeding the mean time to failure of their underlying hardware.

Existing work on shared memory fault tolerance has focused on a variety of system-level solutions. On the hardware side are systems like SafetyNet[3], which insert additional hardware into the system to track data as it moves across the network. This ability allows SafetyNet to efficiently checkpoint system state and roll back to it on failure. Typical software solutions such as [2] embed themselves into the shared memory coherence protocol. This allows them to easily determine where all the pages of shared memory are at any given time, an ability that they use to checkpoint the local and shared state of the system.

While both approaches have had success in showing the practicality of shared memory fault tolerance, their position as system-level tools limits their applicability and scalability in many real-world situations. In particular,

- With no knowledge of the application's source code these tools can do little to minimize the amount of state they save. This ability will become critical as we move to larger shared memory machines and machines made up of clusters of SMPs (like the Earth Simulator), both of which may have a lot of RAM.
- A system-level solution is by definition intimately tied to the operating system/architecture. As a result, such solutions are not portable across platforms and shared memory implementations. This limits users in that it ties them to a particular system configuration.

II. Our approach

It is to address these issues of portability and scalability that we propose our system for application-level checkpointing of OpenMP programs. The idea behind application-level checkpointing is to modify the source code of the program so that it can save just the key problem state rather than the entire system state. If some failure occurs, the system will roll back to a recently saved state and continue computing as normal.

The goal of our system is to automate application-level checkpointing for OpenMP programs written in C by providing a precompiler tool that can modify the application to save its own state. A key feature of this tool is that it generates consistent network-wide snapshots of the shared memory with no knowledge of the underlying OpenMP implementation.

III. Checkpointing Algorithm

In application-level checkpointing potential checkpoint locations are placed statically into the program's source code. The checkpointer's job is twofold: 1. save all shared and local state and 2. ensure that the saved state is consistent across the network.

Our system initiates a global checkpoint by picking a potential checkpoint location within each thread. When a thread reaches its respective checkpoint location, it synchronizes on a barrier. Once every thread has reached this barrier, every thread moves on to save its local state. Working at the application level, we have no direct knowledge of which processor has which portion of the shared memory. Therefore, we have thread 0 save all the shared state.

Overall, the state saved includes the global variables, the heap, the call stack, the local variables and the program counter and our state saving mechanism is similar to that used by [1], with important modifications required to handle OpenMP state and directives. In particular our use of barriers to synchronize checkpointing can interfere with the application's use of barriers and locks, a problem we've had to develop special protocols to deal with.

We have performed 16 processor performance tests on an early version of our implementation. For realistic problem sizes and a checkpoint interval of 45 seconds (much more frequent than in realistic usage), the overhead of our solution ranged between 1% and 5%.

IV. References

1. Bronevetsky et al. "Automated Application-level Checkpointing of MPI Programs". PPoPP 03.
2. Kongmunvattana et al. "Coherence-Based Coordinated Checkpointing for Software Distributed Shared Memory Systems". ICDCS 00.
3. Sorin et al. "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery". ISCA 02.