PORTABLE CHECKPOINTING FOR HETEROGENEOUS ARCHITECTURES

Volker Strumpen*
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139, USA
strumpen@theory.lcs.mit.edu

Balkrishna Ramkumar
Centaur Technologies, Inc.
Iowa City, IA 52242, USA
centaur@avalon.net

Abstract: Current approaches for checkpointing assume system homogeneity, where checkpointing and recovery are both performed on the same processor architecture and operating system configuration. Sometimes it is desirable or necessary to recover a failed computation on a different processor architecture. For such situations checkpointing and recovery must be portable. We argue that source-to-source compilation is an appropriate concept for this purpose. We describe the compilation techniques developed for the design of the c2ftc prototype, which enables machine-independent checkpoints by automatic generation of checkpointing and recovery code. Sequential C programs are compiled into fault tolerant C programs, whose checkpoints can be migrated across heterogeneous networks, and restarted on binary incompatible architectures. Experimental results on several systems provide evidence that the performance penalty of portable checkpointing is negligible for realistic checkpointing frequencies.1

*This work was supported in part by DARPA Grant N00014-94-1-0985.
1A preliminary version of this chapter appeared in the Digest of Papers of the 27th International Symposium on Fault-Tolerant Computing (Ramkumar and Strumpen, 1997).
INTRODUCTION

Large distributed systems are inherently heterogeneous in nature. Even relatively small local area networks usually consist of a mixture of binary incompatible hardware components and are operated by an even larger variety of operating systems. Providing application fault tolerance in such environments is a key technical challenge, especially since it requires that checkpointing and recovery be portable across the constituent architectures and operating systems.

Obstacles to portability range from the architecture over the operating system to the language level. **Architectural hazards** include differences in representations of basic data types and alignments, or specialized hardware support for programming languages, e.g. register windows of SPARC architectures, that affect the stack frame layout. Differences among implementations of **operating systems**, such as UNIX flavors BSD/System V/Linux, surface in specific process address space layouts and system calls. Several **programming language features** are not portable. Dynamic memory management schemes vary from system to system. Pointers are not portable across different address space layouts. Implementations of low-level features like *setjmp/longjmp* and socket or pipe-based communication mechanisms further exacerbate the problem.

In this chapter, we argue that the two key requirements for portable checkpoints — stack environment portability and pointer portability — can be provided by means of source-to-source compilation based on the following code transformations: (1) Provide machine-independence of the stack environment by compiling entry points into functions that resemble computed goto's. The key to manipulating the program counter and stack pointer in a portable manner is to use the basic function call and return mechanism at the source code level. (2) Pointers are transformed into portable, checkpoint-internal offsets. Other aspects such as data representation conversion, and runtime support are not covered in this chapter, but can be found in (Strumpen and Ramkumar, 1996).

We have developed **c2ftc**, a source-to-source compiler that translates C programs into fault tolerant C programs. This prototype generates code for saving and recovering portable checkpoints to enable fault tolerance across heterogeneous architectures. The **c2ftc** compiler instruments the source program based on potential checkpoint locations in the program specified by the programmer, or inserted by the compiler. These checkpoints can be restored on binary incompatible architectures. **c2ftc** maintains checkpoints in a **Universal Checkpoint Format** (UCF), a machine independent format which is customizable for any given configuration of heterogeneous machines by specifying basic data types including byte order, size and alignment. Other representation issues such as the encoding of denormalized numbers can be handled by supplying architecture specific conversion routines.

The remainder of this chapter is organized as follows. First, we describe related work in the area of checkpointing and recovery. Second, the benefits of the source-to-source compilation approach to portable checkpointing are emphasized. Third, we introduce our approach for checkpointing the runtime