Chapter 9
Dynamic Analysis and Profiling of Multithreaded Systems

Daniel G. Waddington
Lockheed Martin, USA

Nilabja Roy
Vanderbilt University, USA

Douglas C. Schmidt
Vanderbilt University, USA

ABSTRACT

As software-intensive systems become larger, more parallel, and more unpredictable the ability to analyze their behavior is increasingly important. There are two basic approaches to behavioral analysis: static and dynamic. Although static analysis techniques, such as model checking, provide valuable information to software developers and testers, they cannot capture and predict a complete, precise, image of behavior for large-scale systems due to scalability limitations and the inability to model complex external stimuli. This chapter explores four approaches to analyzing the behavior of software systems via dynamic analysis: compiler-based instrumentation, operating system and middleware profiling, virtual machine profiling, and hardware-based profiling. We highlight the advantages and disadvantages of each approach with respect to measuring the performance of multithreaded systems and demonstrate how these approaches can be applied in practice.

INTRODUCTION

Microprocessors execute code as a sequential flow of instructions. Most contemporary operating systems support multitasking, which allows more than one program to execute simultaneously. Multitasking is achieved by dynamically scheduling different executions to the available processors over time (sometimes referred to as time slicing).
The unit of logical flow within a running program is a thread. Although the exact definition of a thread can vary, threads are typically defined as a lightweight representation of execution state. The underlying kernel data structure for a thread includes the address of the run-time stacks, priority information, and scheduling status. Each thread belongs to a single process (a process requires at least one thread). Processes define initial code and data, a private virtual address space, and state relevant to active system resources (e.g., files and semaphores). Threads that belong to the same process share the same virtual address space and other system resources. There is no memory protection between threads in the same process, which makes it easy to exchange data efficiently between threads. At the same time, however, threads can write to many parts of the process’ memory. Data integrity can be quickly lost, therefore, if access to shared data by individual threads is not controlled carefully.

Threads have traditionally been used on single processor systems to help programmers implement logically concurrent tasks and manage multiple activities within the same program (Rinard, 2001). For example, a program that handles both GUI events and performs network I/O could be implemented with two separate threads that run within the same process. Here the use of threads avoids the need to “poll” for GUI and packet I/O events. It also avoids the need to adjust priorities and preempt running tasks, which is instead performed by the operating system’s scheduler.

With the recent advent of multicore and symmetric multiprocessor (SMP) systems, threads represent logically concurrent program functions that can be mapped to physically parallel processing hardware. For example, a program deployed on a four-way multicore processor must provide at least four independent tasks to fully exploit the available resources (of course it may not get a chance to use all of the processing cores if they are occupied by higher priority tasks). As parallel processing capabilities in commodity hardware grow, the need for multithreaded programming has increased because explicit design of parallelism in software is now key to exploiting performance capabilities in next-generation processors (Sutter, 2005).

This chapter reviews key techniques and methodologies that can be used to collect thread-behavior information from running systems. We highlight the strengths and weaknesses of each technique and lend insight into how they can be applied from a practical perspective.

**Understanding Multithreaded System Behavior**

Building large-scale software systems is both an art and an engineering discipline. Software construction is an inherently iterative process, where system architects and developers iterate between problem understanding and realization of the solution. A superficial understanding of behavior is often insufficient for production systems, particularly mission-critical systems where performance is tightly coupled to variations in the execution environment, such as load on shared resources and hardware clock speeds. Such variations are common in multithreaded systems where execution is affected directly by resource contention arising from other programs executing at the same time on the same platform. To build predictable and optimized large-scale multithreaded systems, therefore, we need tools that can help improve understanding of software subsystems and help avoid potential chaotic effects that may arise from their broader integration into systems.

Multithreaded programs are inherently complex for several reasons (Lee, 2006; Sutter & Larus, 2005), including: (1) the use of nondeterministic thread scheduling and pre-emption; and (2) control and data dependencies across threads. Most commercial-off-the-shelf (COTS) operating systems use priority queue-based, preemptive thread scheduling. The time and space resources