SOSP17, Work in progress session, December 14, 1999.

Developing Correct and Efficient Multithreaded Programs with Thread-Specific Data and a Partial Evaluator

Yasushi Shinjo
Institute of Information Sciences and Electronics
University of Tsukuba
http://www.is.tsukuba.ac.jp/~yas/

Calton Pu
College of Computing
Georgia Institute of Technology
http://www.cc.gatech.edu/~calton/

1 Introduction

Multithreaded programming is difficult and it requires a higher skill than single threaded programming. In this paper, we describe a development method of correct and efficient multithreaded programs using *thread-specific data* (TSD) and a partial evaluator. Using TSD is a common technique to develop thread-safe programs. It is much easier to write a correct program with TSD than to do with synchronization primitives, such as locking. In SMPs (symmetric multiprocessors), programs with TSD are potentially scalable because no locking is required and caching works well.

The problem of TSD is performance. A TSD reference in the POSIX Threads Standard requires a function call and several tens of instructions while a simple global reference requires just one or two instructions. Therefore, TSD has been considered expensive when it is compared to global data.

To solve this problem, we use runtime specialization and code generation techniques. We convert a TSD reference into a simple global reference automatically by using a partial evaluator. This result changes multithread programmers' mind drastically. Since TSD has been considered expensive, TSD is used carefully and function calling for TSD is minimized in existing programs. From now on, multithread programmers can use TSD not only for developing correct programs easily but also for achieving scalability in SMPs.

2 Specializing a random number generator with TSD

We use a random number generator as an application program. Faster pseudo random number generators are demanded by many simulation applications and encryption applications. We choose this application to show that our method can be applied for fine-grained usage of TSD. GNU libc version 2 includes a simple and fast random number generator with a reentrant interface. This function takes an extra pointer to save intermediate states. We have modified this function to use the TSD facility in POSIX.

Figure 1 shows how to get a specialized random number generator by Tempo[1], a partial evaluator for the C language. Tempo takes a source program and hints in C and ML. In this figure, the variable "**randkey**" is described as an invariant. From this hint, Tempo can reason that the return value of the function **pthread_getspecific()** is also invariant. Tempo finally produces a template and a runtime specializer in C. In the template, the function call for TSD is replaced with a *hole*, the address of a

[1] COMPOSE Project. http://www.irisa.fr/compose/

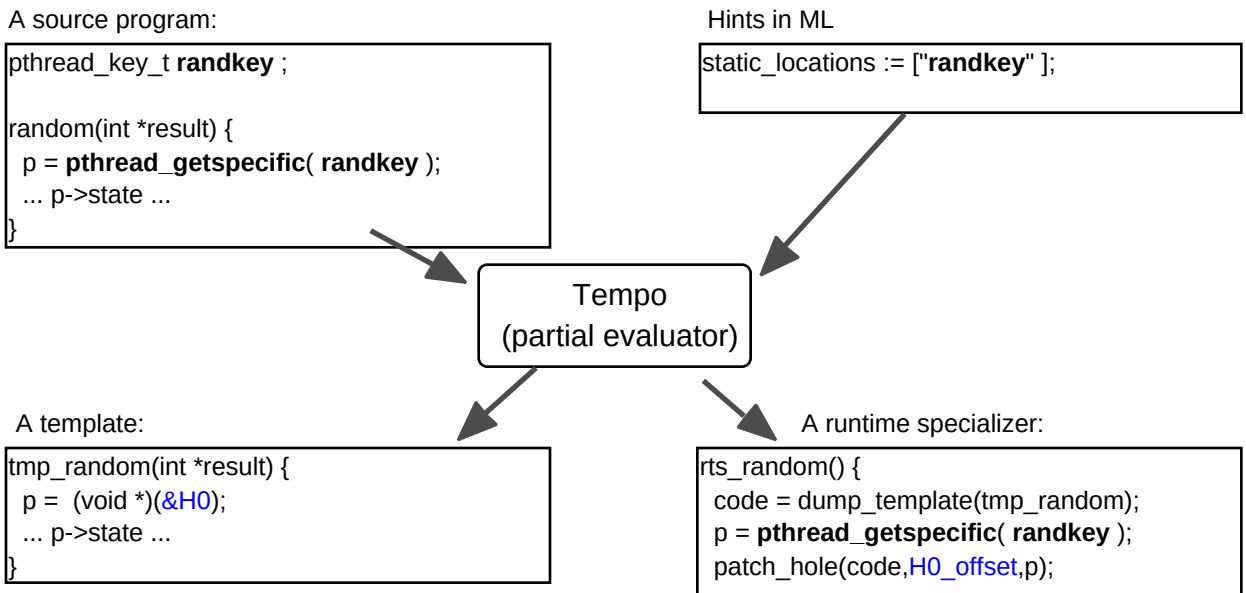


Figure 1: Specializing a random number generator by Tempo

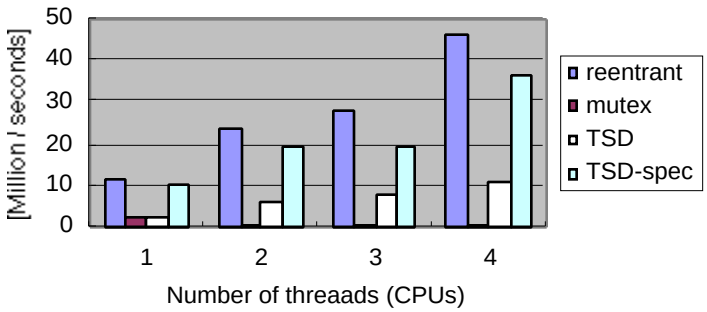
dummy external variable. When the runtime specializer is called, it copies the template, and fills the hole with the return value of the function **pthread_getspecific()**. As a result, the generated code does not include the function call but includes one or two instructions to produce the return value.

3 Experiments

Figure 2 shows throughputs of random number generators. The x-axis is the number of worker threads and the number of requested CPUs for the operating system. The y-axis is throughput in million random numbers per second. This means how many random numbers are generated in a second. Each execution of the benchmark program produces 1 to 4 million of random numbers. Each program is executed repeatedly 100 times and its average throughput is shown in Figure 2.

In this experiment, the mutex function is not scalable because this benchmark does not include enough parallelism to consume random numbers and caching does not work well. The other programs are fairly scalable. In this environment, the function with the reentrant interface is best. Although the TSD function is scalable, it is much slower than the reentrant function. The specialization improves performance by a factor of 3 (compare TSD and TSD-spec in Figure 2), and the specialized function is comparable with the reentrant function.

Figure 2: Throughput of random number generators



The experimental environment is as follows:

Machine: Sun Enterprise 450
OS: Solaris 7 (SunOS 5.7)
CPU: 4 x UltraSPARC II 296MHz
External Cache 2MB.
C Compiler: gcc version 2.95.2
Options: -O2 -fno-schedule-insns
Algorithm: a linear feedback shift register,degree 31