A Determinizing Compiler

Nalini Vasudevan
Columbia University, New York
naliniv@cs.columbia.edu

Stephen A. Edwards
Columbia University, New York
sedwards@cs.columbia.edu

Abstract

The advent of multicores mandates parallel programming. While parallelism presents a panoply of problems, few are as pernicious and prevalent as nondeterminism, in which the output of a program is affected by more than just its inputs, e.g., uncontrollable scheduling choices made by the operating system. A few parallel languages do guarantee determinism, but do so through draconian restrictions.

It is time for a new era of bug-free parallel programming that will enable programmers to shift easily from sequential to parallel worlds. We propose a determinizing compiler: starting from a non-deterministic program, our compiler inserts just enough additional synchronization to guarantee deterministic behavior, even in the presence of nondeterministic scheduling choices. A brute-force solution would simply generate sequential code, but our compiler will strive to preserve parallelism to impose a minimal loss of performance.

1. The Problem

Consider the C program in Figure 1, which creates two threads using *pthread_create()*. The two threads execute *foo()* and *bar()* concurrently, then *pthread_join()* calls wait for both threads to finish and *x* is printed. *Qux()* and *baz()* are functions that do not share any variables. This program has a data race because *x* is being read and modified by two concurrent tasks: *foo()* and *bar()*.

Figure 2 is a modification of Figure 1 that encloses accesses *x* within locks. This prevents races, but the program is still non-deterministic: the final value of *x* depends on the schedule. Suppose *m* evaluates to 1 and *n* evaluates to 2. If *foo* modifies *x* first, the printed value of *x* is $(1+1)*2=4$ because *x* is initialized to 1. However, if *bar* modifies *x* first, the value printed is $(1*2)+1=3$. The problem is that the functions foo and *bar* do not commute.

On a two-processor machine (an Intel Core 2 Duo running Windows XP), we ran this program ten times and it consistently printed 4. We then ran the program on a different machine (a Pentium 4 running Linux), and the program printed 3 six times and 4 four times. Such non-deterministic behavior is undesirable and can be a serious problem in safety-critical applications.

Deterministic programming languages and models such as Kahn's [3], StreamIt [6], and SHIM [1] do not allow such behavior, guaranteeing determinism in part by eschewing shared memory. However, they make programming difficult by imposing too many restrictions on the behavior and structure of programs.

```
int x = 1;
void *foo(void *args) {
  int m;
  m = qux();
  x = x + m;
}
void *bar(void *args) {
  int n;
  n = baz();
  x = x * n;
}
main() {
  ..
  pthread_create(&t_bar, NULL, bar, NULL);
  pthread_create(&t_foo, NULL, foo, NULL);
  ..
  pthread_join(t_bar, NULL);
  pthread_join(t_foo, NULL);
  printf("%d", x);
}
```

Figure 1. A program with races

```
int x = 1;
void *foo(void *args){
  int m;
  m = qux();
  pthread_mutex_lock(&mutex);
  x = x + m;
  pthread_mutex_unlock(&mutex);
}
void *bar(void *args){
  int n;
  n = baz();
  pthread_mutex_lock(&mutex);
  x = x * n;
  pthread_mutex_unlock(&mutex);
}
main() {
  ..
  pthread_create(&t_bar, NULL, bar, NULL);
  pthread_create(&t_foo, NULL, foo, NULL);
  ..
  pthread_join(t_bar, NULL);
  pthread_join(t_foo, NULL);
  printf("%d", x);
}
```

Figure 2. A race-free, non-deterministic program

```
int x = 1;
void *foo(void *args) {
  int m;
  m = qux();
  x = x + m;
  sync(x); /* Wait for bar to sync */
}
void *bar(void *args) {
  int n;
  n = baz();
  sync(x); /* Wait for foo to sync */
  x = x * n;
}
main() {
  // ..
  pthread_create(&t_bar, NULL, bar, NULL);
  pthread_create(&t_foo, NULL, foo, NULL);
  // ..
  pthread_join(t_bar, NULL);
  pthread_join(t_foo, NULL);
  printf("%d", x);
}
```

Figure 3. A deterministic version of the program in Figure 1

2. The Solution

We propose a radical change to parallel programming: programmers will have the freedom to introduce bugs, but still manage to obtain correct executions. We propose a compiler that will automatically generate deterministic code from a non-deterministic program. The generated code will be scheduling-agnostic and will thereby produce reproducible behavior.

We our compiler will generate code like Figure 3 for the program in Figure 1. This version forces the two tasks to synchronize (rendezvous) at the *sync(x)* statement. The *sync* statement acts as a barrier that forces a partial order between the two tasks that orders accesses to *x* but allows the functions *qux()* and *baz()* to run concurrently. The program in Figure 3 is scheduler-agnostic: if *m* is 1 and *n* is 2, it always prints 4.

Our idealistic goal is to take any C program with parallel constructs and generate deterministic parallel code. The main issue is to find out if a pair of tasks commute. We will use existing race-detection tools [5, 2] as a starting point. Although not all race-free programs are deterministic, we plan to use concepts from race-detection algorithms in our implementation. Then, we will tune the compiler for optimal performance (i.e., permitting the largest amount of parallelism that does not violate scheduling independence) by performing timing analysis [7, 4]. We will also consider machine learning algorithms to assist the compiler in selecting the programmer-intended execution among the different non-deterministic executions.

There are many open challenges to the design and implementation of such a compiler. How successful can we be with only an approximate analysis of a program? How do we deal with pointers? Where, exactly, do we insert synchronization? Although there are a few solutions that already exist for some of these sub-problems, they are not complete. We will pick the appropriate ones, modify and integrate them to suit our purpose. We also plan to use approximation and compositional techniques to tackle these problems. Our ultimate goal is to strike the perfect balance between determinism and parallelism.

One way to approach this problem would be to convert the program into a completely sequential one (i.e., force a total order over all instructions) then selectively remove synchronization constraints that we can show to be unnecessary (such as any restrictions on the order between the executions of *qux* and *baz* in our running example). This technique might avoid a problem noted by one of our reviewers: how consistently our compiler will determinize a particular program. While our determinizing algorithm itself will certainly be deterministic, there is some danger that it might be sensitive to small changes in the source code. However, it should be straightforward to develop a sequentialization algorithm (using, e.g., program source layout as a hint) that would be insensitive to small modifications.

Going forward, we expect every compiler will be determinizing as well as optimizing. We believe this will be a necessary step along the way to pervasive parallelism in programming.

References

[1] Stephen A. Edwards. SHIM: A language for hardware/software integration. In *Proceedings of SYNCHRON*, Schloss Dagstuhl, Germany, December 2004.

[2] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks, 2003.

[3] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[4] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Proceedings of the 34th Design Automation Conference*, June 1997.

[5] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[6] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196, Grenoble, France, April 2002.

[7] Reinhard Wilhelm. Timing analysis and timing predictability. In *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323, Leiden, The Netherlands, November 2004.