ECE459: Programming for Performance Winter 2011

Lecture 16 — March 8, 2011

Patrick Lam *version 1*

Program note: We slipped a lecture since the OpenMP lecture notes (L13) covered two lectures, so I'm re-aligning the lecture numbers with dates here.

This week will focus on compiler optimizations. But first, some related topics. A general reference and source of examples for this lecture is:

Hans-J. Boehm. "Threads cannot be implemented as a library". Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 261–268. `http://portal.acm.org/citation.cfm?doid=1065010.1065042`

Memory Consistency, Memory Barriers, and Reordering

The L14 notes (once I finish them) explain how to specify memory barriers in OpenMP. Today we'll talk a bit more about memory consistency, memory barriers and reordering in general. In this part of the lecture, we mean instruction reordering by the CPU; in the second part of the lecture, we'll mean reordering initiated by the compiler.

Memory Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model. What are the possible values for the variables?

It turns out that sequential consistency is too expensive to implement. (Why?) So most systems actually implement weaker memory models, such that both `r1` and `r2` might end up unchanged.

Reordering. Compilers and processors may reorder non-interfering memory operations within a thread. For instance, the two statements in `T1` appear to be independent, so it's OK to execute them—or, equivalently, to publish their results to other threads—in either order. Reordering is one of the major tools that compilers use to speed up code.

When is reordering a problem?

Memory Barriers. We previously talked about OpenMP barriers: at a `#pragma omp barrier`, all threads pause, until all of the threads reach the barrier.

A rather different type of barrier is a *memory barrier* or *fence*. This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occuring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible. The x86 architecture defines the following types of memory barriers:

- `mfence`. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- `sfence`. All stores before the barrier become visible before all stores after the barrier become visible.
- `lfence`. All loads before the barrier become visible before all loads after the barrier become visible.

You can use the `mfence` instruction to implement *acquire barriers* and *release barriers*. An acquire barrier ensures that memory operations after a thread obtains the mutex doesn't become visible until after the thread actually obtains the mutex. The release barrier similarly ensures that accesses before the mutex release don't get reordered to after the mutex release. Note that it is safe to reorder accesses after the mutex release and put them before the release.

Here's an example spinlock implementation which uses barriers (Listing 8.7 in Gove):

```
void lock_spinlock(volatile int *lock) {
  while (CAS(lock, 0, 1) != 0) {} // compare-and-swap
  acquire_memory_barrier();
}

void free_spinlock(volatile int *lock) {
  release_memory_barrier();
  *lock = 0;
}
```

`volatile`. This qualifier ensures that the code does an actual read from `lock` every time it asks for one (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Compiler Optimizations

The main plan for this week is to talk about compiler optimizations. We'll talk about general optimizations (which are definitely performance-related) as well as the complications that you encounter with these optimizations in the presence of threads. One particular type of optimizations we'll talk about is feedback-directed optimizations.

C compilers contain a rich suite of optimization options[1][2]. The simplest set of optimizations are low-level optimizations, which the compiler uses in the final translation into assembly code. Choosing the appropriate instruction set can be a win here: you can tell the compiler to emit 64-bit instructions, MMX, SSE, SSE2, etc. Solaris Studio uses `-xarch` and `-xchip`, while `gcc` uses `-march` and `-mtune`. *Peephole optimizations* are also low-level optimizations; they replace more-expensive sequences of instructions with less-expensive sequences. For instance, it was historically a win to replace `* 2` by `<< 1`.

Compilers usually have a general optimization flag; Solaris uses `-xOn`, where n is between 0 and 5, while `gcc` uses `-On`, where n is between 0 and 3. 0 means no optimization.

Solaris Studio. Let's investigate the Solaris optimizations. `-fast` is a suite of common optimizations:

```
-fns -fsimple=2 -fsingle -nofstore -xalias_level=basic -xbuiltin=%all
-xdepend -xlibmil -xlibmopt -xO5 -xregs=frameptr -xtarget=native
```

It includes some floating-point options which may be incorrect in some cases (i.e. does not meet IEEE 754), but should be fine most of the time. It also assumes that the program respects C basic types when dereferencing pointers, i.e. you never access an `int` through a `char *` pointer. (The only correct way of doing that in C is through a `union`.) More details about that next lecture, under my discussion of "alias analysis". It also tells the compiler to assume that you never redefine library functions (as in my earlier `sin()` example) and lets it use optimized library and math routines.

Inlining. Function calls are slow and prevent further optimization. One of the most useful optimizations is therefore inlining: copying the code of the a function into its caller.

Why is inlining hard? What are the tradeoffs in deciding whether to inline?

Basic optimizations. Here is a list of the "basic" optimizations that the Solaris Studio compiler performs on SPARC processors under `-xO2`:

induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, complex expression expansion

An example of common subexpression elimination is that instead of computing `x + 2` twice in a method, you compute it once, store it, and use the stored copy next time.

[1]`http://developers.sun.com/solaris/articles/amdopt.html`
[2]`http://developers.sun.com/solaris/articles/options.html`