Usenet Nuggets

by Mark Thorson
mmm@cup.portal.com

This column consists of selected traffic from the *comp.arch* newsgroup, a forum for discussion of computer architecture on Usenet—the international network of Unix-based computers.

As always, the opinions expressed in this column are the personal views of the authors, and do not necessarily represent the institutions to which they are affiliated.

Text which sets the context of a message appears in italics; this is usually text the author has quoted from earlier Usenet traffic. The code-like expressions below the authors' names are their addresses on Usenet.

Stack Machines

Phil Koopman
koopman@utrcgw.utc.com

This is a consolidated attempt to address some of the points people have been bringing up about stack-based architectures. It's written from the point of view of someone who has actually made a living designing and selling them (first with WISC Technologies, later with Harris Semiconductor RTX family).

Pipelining

Stack processors don't need to be pipelined for ALU and operands, because the operands are immediately available in the top of stack buffer registers. Access to the on-CPU stack RAM can be completely hidden by pipelining. Access to off-chip memory is typically not pipelined in current implementations, but can be (P. Koopman, *Some Ideas for Stack Computer Design*, 1991 Rochester Forth Conference, pg. 58). It makes sense that the false dependencies introduced by stack addressing could be overcome with a superscalar implementation if one were so inclined (but, I'm not).

Stack Size and Interrupts

On-chip stack buffers only need to be about 16 deep. Spilling can be done by stack overflow interrupts (or, for that matter, statically scheduled by the compiler the same as register spills). Cost for interrupt-driven overflows is less than 1% for only 16 registers and essentially 0% for 32 registers for reasonable programs (P. Koopman, *Stack Computers*, pp. 139–146).

A neat thing about stack CPUs is that context switching for interrupts takes essentially zero time—no registers need to be saved; you just put the ISR values onto the top of the stack and clean them off when you're done (presumes enough stack space is available—no big deal to arrange). (P. Koopman, *Stack Computers*, pp. 146-152.)

Program Size

Program size doesn't matter (much) for workstations. But, for embedded control it matters a lot, especially when you're limited to on-CPU chip memory, and the CPU has to cost less than $5–$10. Anecdotal evidence indicates stack computer program size can be smaller than CISC programs by a factor of 2.5 to 8 (and, another factor of 1.5 to 2.5 smaller than RISC, depending whom you want to believe). This comes not just from compact opcodes, but also from reuse of short code segments and implicit argument passing with subroutine calls. Code size comparisons I've seen don't take this into account. (P. Koopman, *Stack Computers*, pg. 118-121.)

Compilers

Stack compilers aren't currently very efficient—but that's because no-one has really tried all that hard. I've published an algorithm and experimental results that suggest that stacks can be made about as efficient as registers in terms of keeping local variables at the top level of the memory hierarchy. The work is based on GNU C intermediate code (P. Koopman, *A Preliminary Exploration of Optimized Stack Code Generation*, 1992 Rochester Forth Conference, in press; uuencoded postscript copy of paper available upon request).

I'm currently working (at hobby level) on a GNU C stack-based compiler that will generate very compact code. Of course, one could always argue that trying to map C to a stack machine will necessarily be less efficient than mapping a stack-friendly language such as Forth to a stack machine. The issue starts to have more to do with marketing than technology, but seems like a neat challenge.

Applications

Overall, I'd say stack machines are now an excellent fit for high performance in a low cost system (not necessarily highest performance given unlimited cost). They should do especially well in embedded applications.

Cache Size and Garbage Collection

Paul R. Wilson
wilson@cs.utexas.edu

Since I'll have some money to spend real soon now, I've been pondering whether we should switch from Sun to HP. The usual benchmark results suggest that a change might result in faster execution of our pet program, a large "theorem prover" written in Lisp. From the SpecInt92 results, one should expect a performance increase of about a factor of 2 when switching from a SS2 to either a SS10-30 or HP720. Unfortunately, both machines turned out to be just 30% faster than a SS2 when running our system (for which ps never shows a resident set under 3–4 MB, not even on an 8MB machine). Any explanations at hand? Did we encounter a bottleneck between CPU and memory, or what?

Could be. If you don't have a generational garbage collector (GC), your locality is going to be the pits. If you allocate a lot of data between garbage collections, you'll typically incur a cache miss and a writeback for every block of memory you allocate. That's because you can't reuse memory until you know it's garbage, so you're always allocating something you haven't used for a long time, i.e., at least since the last garbage collection.

What you want is a generational garbage collector and a cache large enough to hold the youngest generation. This lets you allocate less-than-a-cache-full of data between garbage collections, reclaim most of the space, and reuse it at the next GC cycle.

The youngest generation should generally be >100KB for basic GC efficiency reasons (space-time tradeoffs), so you can't really expect to stay in a first-level cache for your heap allocations. You could stay in a megabyte-range second or third level cache.

What performance should I expect from a SS10-41 or SS10-52 (which have a larger cache, but still not large enough to hold the Lisp's resident set. And if I'm not mistaken, the large cache results in a longer time spent for non-cache memory accesses [6 cycles instead of 3]). What performance should I expect from an HP735?

If cache misses on allocation are your problem, you're limited by the rate of allocation and the cache miss service time, plus something for write backs. (You'll typically incur a write-back of a dirty block for each block of heap data you allocate, since the cache will be mostly full of relatively recently allocated—hence written—garbage. This can overload your write buffers in a hurry for some programs.)

You also need to add something extra if it's a direct-mapped cache—GC'ed systems are especially sensitive to DM cache conflicts. (Actually, it's kinda weird—DM works *better* if the youngest generation almost fits in the cache, but not quite.)

So if you know the rate of allocation in your application, you should be able to figure a ballpark cache miss cost without much trouble.

(For more on this, see Wilson, Lam, and Moher, *Caching Considerations for Generational Garbage Collection*, ACM Lisp and Functional Programming '92.)

I can buy enough memory so that disk speed is no criterion. Should I wait for machines with 8MB of cache? Should I shoot myself? Should I give you the money? :-)