Modifying the 'middle end' of a popular compiler yields more-efficient parallel programs

January 30 2017, by Larry Hardesty

“Everybody said it was going to be too hard, that you’d have to change the whole compiler,” MIT professor Charles E. Leiserson says. “And these guys basically showed that conventional wisdom to be flat-out wrong.” Credit: Massachusetts Institute of Technology

Compilers are programs that convert computer code written in high-level languages intelligible to humans into low-level instructions executable by machines.

But there's more than one way to implement a given computation, and modern compilers extensively analyze the code they process, trying to deduce the implementations that will maximize the efficiency of the resulting software.

Code explicitly written to take advantage of parallel computing, however, usually loses the benefit of compilers' optimization strategies. That's because managing parallel execution requires a lot of extra code, and existing compilers add it before the optimizations occur. The optimizers aren't sure how to interpret the new code, so they don't try to improve its performance.

At the Association for Computing Machinery's Symposium on Principles and Practice of Parallel Programming next week, researchers from MIT's Computer Science and Artificial Intelligence Laboratory will present a new variation on a popular open-source compiler that optimizes before adding the code necessary for parallel execution.

As a consequence, says Charles E. Leiserson, the Edwin Sibley Webster Professor in Electrical Engineering and Computer Science at MIT and a coauthor on the new paper, the compiler "now optimizes parallel code better than any commercial or open-source compiler, and it also compiles where some of these other compilers don't."

That improvement comes purely from optimization strategies that were already part of the compiler the researchers modified, which was designed to compile conventional, serial programs. The researchers' approach should also make it much more straightforward to add optimizations specifically tailored to parallel programs. And that will be

crucial as computer chips add more and more "cores," or parallel processing units, in the years ahead.

The idea of optimizing before adding the extra code required by parallel processing has been around for decades. But "compiler developers were skeptical that this could be done," Leiserson says.

"Everybody said it was going to be too hard, that you'd have to change the whole compiler. And these guys," he says, referring to Tao B. Schardl, a postdoc in Leiserson's group, and William S. Moses, an undergraduate double major in electrical engineering and computer science and physics, "basically showed that conventional wisdom to be flat-out wrong. The big surprise was that this didn't require rewriting the 80-plus compiler passes that do either analysis or optimization. T.B. and Billy did it by modifying 6,000 lines of a 4-million-line code base."

Schardl, who earned his PhD in electrical engineering and computer science (EECS) from MIT, with Leiserson as his advisor, before rejoining Leiserson's group as a postdoc, and Moses, who will graduate next spring after only three years, with a master's in EECS to boot, share authorship on the paper with Leiserson.

Forks and joins

A typical compiler has three components: the front end, which is tailored to a specific programming language; the back end, which is tailored to a specific chip design; and what computer scientists oxymoronically call the middle end, which uses an "intermediate representation," compatible with many different front and back ends, to describe computations. In a standard, serial compiler, optimization happens in the middle end.

The researchers' chief innovation is an intermediate representation that employs a so-called fork-join model of parallelism: At various points, a

program may fork, or branch out into operations that can be performed in parallel; later, the branches join back together, and the program executes serially until the next fork.

In the current version of the compiler, the front end is tailored to a fork-join language called Cilk, pronounced "silk" but spelled with a C because it extends the C programming language. Cilk was a particularly congenial choice because it was developed by Leiserson's group—although its commercial implementation is now owned and maintained by Intel. But the researchers might just as well have built a front end tailored to the popular OpenMP or any other fork-join language.

Cilk adds just two commands to C: "spawn," which initiates a fork, and "sync," which initiates a join. That makes things easy for programmers writing in Cilk but a lot harder for Cilk's developers.

With Cilk, as with other fork-join languages, the responsibility of dividing computations among cores falls to a management program called a runtime. A program written in Cilk, however, must explicitly tell the runtime when to check on the progress of computations and rebalance cores' assignments. To spare programmers from having to track all those runtime invocations themselves, Cilk, like other fork-join languages, leaves them to the compiler.

All previous compilers for fork-join languages are adaptations of serial compilers and add the runtime invocations in the front end, before translating a program into an intermediate representation, and thus before optimization. In their paper, the researchers give an example of what that entails. Seven concise lines of Cilk code, which compute a specified term in the Fibonacci series, require the compiler to add another 17 lines of runtime invocations. The middle end, designed for serial code, has no idea what to make of those extra 17 lines and throws

up its hands.

The only alternative to adding the runtime invocations in the front end, however, seemed to be rewriting all the middle-end optimization algorithms to accommodate the fork-join model. And to many—including Leiserson, when his group was designing its first Cilk compilers—that seemed too daunting.

Schardl and Moses's chief insight was that injecting just a little bit of serialism into the fork-join model would make it much more intelligible to existing compilers' optimization algorithms. Where Cilk adds two basic commands to C, the MIT researchers' intermediate representation adds three to a compiler's middle end: detach, reattach, and sync.

The detach command is essentially the equivalent of Cilk's spawn command. But reattach commands specify the order in which the results of parallel tasks must be recombined. That simple adjustment makes fork-join code look enough like serial code that many of a serial compiler's optimization algorithms will work on it without modification, while the rest need only minor alterations.

Indeed, of the new code that Schardl and Moses wrote, more than half was the addition of runtime invocations, which existing fork-join compilers add in the front end, anyway. Another 900 lines were required just to define the new commands, detach, reattach, and sync. Only about 2,000 lines of code were actual modifications of analysis and optimization algorithms.

Payoff

To test their system, the researchers built two different versions of the popular open-source compiler LLVM. In one, they left the middle end alone but modified the front end to add Cilk runtime invocations; in the

other, they left the front end alone but implemented their fork-join intermediate representation in the middle end, adding the runtime invocations only after optimization.

Then they compiled 20 Cilk programs on both. For 17 of the 20 programs, the compiler using the new intermediate representation yielded more efficient software, with gains of 10 to 25 percent for a third of them. On the programs where the new compiler yielded less efficient software, the falloff was less than 2 percent.

"For the last 10 years, all machines have had multicores in them," says Guy Blelloch, a professor of computer science at Carnegie Mellon University. "Before that, there was a huge amount of work on infrastructure for sequential compilers and sequential debuggers and everything. When multicore hit, the easiest thing to do was just to add libraries [of reusable blocks of code] on top of existing infrastructure. The next step was to have the front end of the compiler put the library calls in for you."

"What Charles and his students have been doing is actually putting it deep down into the compiler so that the compiler can do optimization on the things that have to do with parallelism," Blelloch says. "That's a needed step. It should have been done many years ago. It's not clear at this point how much benefit you'll gain, but presumably you could do a lot of optimizations that weren't possible."

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology

Citation: Modifying the 'middle end' of a popular compiler yields more-efficient parallel

programs (2017, January 30) retrieved 1 May 2024 from https://phys.org/news/2017-01-middle-popular-yields-more-efficient-parallel.html