PARALLELIZING SEQUENTIAL CODE

Program parallelism is no longer optional in obtaining performance. In the middle of the last decade, processor speeds stopped increasing at the decades-old exponential rate. Instead, the number of processing cores per chip has grown exponentially. There’s no free lunch anymore; only programs with sufficient parallelism benefit from each new hardware generation.

To bridge the gap between parallel hardware resources and performance, programmers must write (or rewrite) applications with the target parallel architecture in mind. This is a serious problem. Practically, this is expensive. Writing parallel programs is almost universally acknowledged to be much harder than writing sequential ones. Doing so requires programmers with a higher level of skill. Even with this skill, programmers will spend a growing fraction of their time on performance tuning and correctness debugging. Requiring knowledge of the implementation of the target parallel architecture limits reuse and makes computing less accessible. Fundamentally, abstraction, a key tenet of computer science, is broken.

One approach to reducing the expense of parallel programming is the automatic parallelization of sequential code. A historical inspiration for this approach is the success of instruction-level parallelism (ILP). Computer architects and compiler writers successfully hid the implementation details, and even the very existence, of ILP hardware. All that programmers and users experienced during this golden age of computer architecture was an immediate increase in performance with each new product generation. Many believe that something similar can happen again at the coarser levels of granularity necessary for the efficient utilization of multicore.

Citing the mixed success of decades of research in the area, many experts are pessimistic about the potential of automatic parallelization. Recent breakthroughs, however, have generated excitement. For example, as speculative techniques improve in efficiency, the need for heroic memory dependence analysis, once thought to be the Achilles’ heel of automatic parallelization, has diminished. Recent work in automatic parallelization targets all types of programs, not just scientific codes. The approaches taken today are radically different from those taken just a decade ago. They may already be sufficient to forever change the way parallel machines are programmed.

As guest editor, I have the pleasure of introducing the IEEE Micro special issue on the parallelization of sequential code. This issue samples progress from the small, but growing, community focused on restoring a fundamental programming abstraction. I hope that these articles will generate increased interest in the automatic parallelization of sequential code and inspire more work in the area.

Automatic extraction of parallelism

The first three articles focus on the automatic extraction of parallel threads from general purpose sequential programs. In each of these articles, we see a different method to identify units of work for multicore processors.

In “Helix: Making the Extraction of Thread-Level Parallelism Mainstream,” Simone Campanoni et al. present developments on an automatically parallelizing compiler based on their Helix approach. This work is a shining example of what parallelization for multicore should be: fully automatic, applicable to a wide range of sequential programs, and functional on commodity

David I. August
Princeton University

Published by the IEEE Computer Society 0272-1732/12/$31.00 © 2012 IEEE
hardware platforms. Given the speedups presented, on the basis of this work alone, one could claim success today for the automatic parallelization approach for multicore systems of up to six cores.

In “Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs,” Feng Li et al. present an interesting approach to the problem of automatic parallelization. They suggest looking at imperative programs as being composed of data-flow threads. With that as a starting point, the natural next step is coarsening these data-flow threads to reduce synchronization and communication overhead. Their early results suggest that a significant amount of concurrency can be extracted by this method. This approach also has the desirable property that it is just as applicable to code with irreducible control-flow graphs and recursive calls.

In “Underclocked Software Prefetching: More Cores, Less Energy,” Md Kamruzzaman et al. parallelize memory-bound codes with helper threads designed to warm the caches with prefetching. Viewed another way, cache misses are serviced concurrently, rather than sequentially, with useful work. The result, when combined with frequency scaling, is reduced energy consumption at each level of performance.

Insight for the future

The last two articles suggest new avenues for progress in the area of automatic parallelization.

“SWAP: Parallelization through Algorithm Substitution,” Hengjie Li et al. advocate a new approach to automatic parallelization. In their system, the compiler substitutes sequential algorithms with their more parallel counterparts using a database of implementations. Parallelizing compilers represent the collected wisdom of their authors, and this article suggests an interesting way in which we might expand this knowledge base.

This special issue demonstrates some of the exciting research going on in the area of automatic parallelization of sequential codes. I hope that it creates discussion, inspires, and informs. I look forward to your feedback on the topics covered in this issue.

Acknowledgments

I am grateful to those who made this issue possible. I thank the authors for their submissions and the reviewers for their careful and thoughtful reviews. I thank my research group for their feedback. Finally, I thank Erik Altman for his invaluable guidance throughout this process.

David I. August is a professor of computer science at Princeton University. His research interests are in synergistic compiler and microarchitecture design. August has a PhD in electrical and computer engineering from the University of Illinois at Urbana-Champaign. He is a member of IEEE and the ACM.

Direct questions and comments about this article to David I. August at august@princeton.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.