Hot Chips and the Incomplete Job of Exploiting Them

Erik R. Altman
Thomas J. Watson Research Center

Welcome to the March/April issue of *IEEE Micro*. As has been the case for decades, we devote this issue to a set of articles describing the top presentations from the most recent Hot Chips conference. Hot Chips continues to introduce many exciting and novel chips, making the task of selecting papers for this issue both a challenge and a feast of riches. Guest Editors Christos Kozyrakis and Rumi Zahir have done an excellent job of recruiting and selecting papers, and I hope that you will enjoy them and learn a lot.

Beyond chips, people often talk about better integrating the software-hardware stack so as to realize performance, efficiency, and energy improvements. Indeed, such ideas have been bandied about for decades, and there are a few successes to show for it, such as RISC-era and resultant closer coupling of compiler and architecture, eventually even for non-RISC architectures. There have been attempts at other cross-stack optimizations, such as network processors and the TCP/IP stack or GPUs and the various languages and APIs—such as OpenGL, CUDA, and OpenCL—that have been developed for their efficient exploitation.

Another example is IBM's Netezza appliance and its optimization of hardware (disk and FPGA) and software for data warehousing. However, as the latter two examples suggest, successes seem to rely more on specialized and deep knowledge of the software application and the hardware than on more general principles that could be broadly applied.

Why are such general principles so hard to develop and deploy? Obvious and much-remarked culprits are modularity and isolation, including the concept of instruction-set architecture (ISA). While these concepts are good for portability of code, speeding development, enhancing security, and providing flexible mix-and-match components, they make it difficult to customize specific combinations of hardware and software to improve performance.

A decade ago, managed runtimes, such as the JVM (Java Virtual Machine), were seen as playing a major role to solve these problems. Such runtimes can see the entire software stack and its dynamic deployment. They can also transparently generate optimized code for specific underlying microarchitectures and ISAs, yielding the proverb, ''write once, run anywhere.'' This slogan has been achieved in significant measure for functionality—but not for performance. Deep call stacks, huge code volumes, and complex intermixings yield a large and exponential set of code paths, and consequently an intractable problem for automatic analyses to determine optimizations across an entire stack: too much memory and too much time are needed.

The problem only grows harder as more concurrency is added, and harder still as system variations accrue: from cache and memory hierarchies to SSD to storage topology to network topology. At the automated-analyses level (for example, at the compiler or JVM), the values for many of these quantities are unknown and thus intractable.

Given this situation, what can we do as microarchitects? Some beginning steps might involve standardization akin to the introduction of ISAs—that is, broaden, standardize, and harden the set of information provided to others, from performance counters, to sampling of the processor/thread state to code line numbers, function names, and module names. Such information exists in bits and pieces but generally is not portable across generations of hardware and software, so it is used only sporadically, not generally.

We might also reconcile ourselves to walking before we run, and more specifically to providing information in a seamless and easy way to experts so that they can understand problems and tweak code and system parameters and microarchitecture. As happened historically with compiler optimizations, the best of the resulting approaches can then be regularized and codified into automatic transformations.

Published by the IEEE Computer Society

Deeper changes may also be appropriate. Traditional information and techniques are almost entirely semantics agnostic—that is, analysis and transformation make no use of the underlying function, and no distinction is made between sorting and matrix multiplication. The top *n* algorithms might be named. Function calls could then be tagged with the *n* algorithms with which they are associated, or they could be tagged with ''other'' if none of the *n* algorithms apply. This tagging information could then be passed to the processor via the ISA or otherwise. If such information were tracked at the microarchitecture level, then at any point in the code, automatic tools, programmers, and performance analysts could understand the range of activities in progress and catch inefficiencies, such as a set of code that does a full sort after every insert operation, instead of one sort after a set of insert operations is completed.

Other techniques are undoubtedly possible as well, and I hope that the microarchitecture community can lead the way in providing standards and information that will indeed let a broad class of workloads be optimized across the stack.

Happy reading!

Erik R. Altman
Editor in Chief
IEEE Micro

Erik R. Altman is the manager of the Dynamic Optimization Group at the Thomas J. Watson Research Center. Contact him at ealtman@us.ibm.com.