RETROSPECTIVE:

Monsoon: An Explicit Token-Store Architecture

Gregory M. Papadopoulos

Sun Microsystems, Inc.
gregp@corp.sun.com

David E. Culler

Computer Science Division
University of California, Berkeley, CA 94720
culler@cs.berkeley.edu

Introduction

Certainly much has changed over the decade since we first prototyped Monsoon. Over that time, dynamic out-of-order instruction execution constrained only by data dependences, with firing rules far more complex than anything we considered practical, became commonplace in microprocessor design. Threads became standard at the operating systems level and incorporated as an intrinsic part of new programming languages. Surely, it would be nice to think of this paper as essential to it all, just ahead of its time. Besides being inaccurate, such a view fails to capture the ideas in this paper that may still have future importance.

Clearly, this paper did not establish dataflow as the dominant principle in instruction set design, nor place functional languages at the center of modern programming. If anything, it was the beginning of the end of dataflow, as it demystified the approach. The Explicit Token Store model established a simple correspondence between dataflow graphs and the spectrum of conventional instruction sets. It provided a clear separation of what should occur above the instruction set level (storage management) and what could occur below it (dynamic instruction scheduling). The Monsoon machine demonstrated that the dataflow firing rule was captured by a simple mechanism: state-dependent instruction completion. In doing so it placed the body of thought associated with dataflow models into a familiar context where its ideas could be more readily harvested.

We decided to structure this retrospective around what we saw to be the four big ideas for the future lurking between the lines. These are outlined in the following sections.

Don't be afraid to build

Monsoon stands as demonstration that a small group of motivated individuals can build complete systems, even from scratch, that differ in fundamental ways from the paths of industry. As a community we seek revolutionary ideas, and these will not come about through incremental variations on well-established techniques, or lighthearted paper studies. You need to live and breathe a new paradigm, and even then it may not come to pass.

Monsoon did work. It was a complete dataflow computing system, in which even the operating system, complete with I/O and storage reclamation, was written in Id and compiled to ETS (Monsoon machine language) code. Routinely, several thousand threads executed concurrently in the machine. A number of systems were built in collaboration with Motorola Cambridge Research Laboratory. Sixteen processor systems were placed at Los Alamos National Laboratory and the MIT Lab for Computer Science, and only recently have they been retired.

A Monsoon processor was able to process 5 to 10 million messages per second. This is still a very respectable rate.

It demonstrated that threads could be dynamically spawned and terminated at this same rate, where threads share registers, as long as the compiler manages the storage in which these threads operate. Many developments in threaded run time systems, including TAM, P-Risc, Filaments, Choros, and Cilk build upon this concept, although recent work has advanced the techniques for managing the scheduling data structures to provide storage guarantees.

We took the dataflow paradigm seriously, lived it, breathed it, and built accordingly. The shortcomings of Monsoon, as well as its successes, were real; they were not artifacts of a "merely academic" investigation.

Evolution of Instruction Sets

One of the clear shortcomings of Monsoon was the power, or lack thereof, of its basic instructions. Conventional instruction sets have evolved over a sequence of steps allowing more and more state to be accessed in each instruction in a single thread. Accumulator based machines gave way to 2-address and 3-address general purpose register instruction sets. We are beginning to look at as many as 128 registers associated with a single thread of control.

Threads are a key agent within all modern machines, and yet there are no operations defined on them at the machine level. Thread operations, such as create and terminate are implemented in software by the operating system and are combined with large storage allocations.

Architects have worked harder and harder to find enough parallelism in a program that has a single control thread to keep the many function units busy; we expand the architected registers through renaming, look far ahead and execute speculatively across even multiple branches. In the window of instructions behind the instruction fetch, execution of many small operations is sequenced dynamically according to data dependences. Effectively, modern microprocessors construct a small window of dynamic dataflow execution on-the-fly.

ETS showed that when dataflow graphs are used as an instruction set, rather than an internal scheduling mechanism, they correspond to a single accumulator architecture that has advanced in the orthogonal direction of allowing many threads to deal with the architected machine state. The accumulator-style of instruction set meant that basic operations in threads — fork and rendezvous — were very efficient, but evaluating simple arithmetic expressions suffered.

Clearly, there is a whole space of designs between these two major axes of evolution: state per operation, and thread expressiveness. We expect that future architectures will support multithreading of fairly stateful instruction sets (e.g., Tera). In this pursuit, we hope that the design point of Monsoon — threads being efficiently virtualized — will not be overlooked because of the lack of power in the operations of individual threads. The authors are still of the belief that threads should be given first-class status and not be viewed merely as a state multiplexing mechanism for latency tolerance.

As we begin to explore the space between the two axes of instruction set evolution, a fundamental question is how efficiently we are able to encode parallelism at the machine level. Both extremes — a single thread with many register names or many threads with few register names — are surely suboptimal.

Program the Memory

One of the important ideas lurking in the Monsoon paper is that of programming the memory. Again, it is useful to look at this in light of the evolution of instruction sets. It used to be that memory references were a piece of an instruction. Typically, an instruction specified an addressing mode with each operand and in many cases the addressing mode could include one or more memory references. In load-store architectures a memory reference is a full instruction. If an instruction is going to access data in memory, it says so right in the opcode and does not try to do anything else. This approach recognizes that memory references are slow and complex relative to arithmetic processing; by calling them out specially it is possible to optimize around them in order to hide latency and resolve dependences. Indeed, today a memory operation involves extremely complex protocols at several levels that far exceed the actual time to access the bits on the memory chip. Issues of when these operations complete, when their effects become visible, and when dependent operations can proceed are quite subtle.

ETS took an important step in treating every memory reference as multiple instructions. Each memory reference was split-phase, so there was an explicit point where the reference was issued and a point where it completed. Given the threaded execution model, it was natural for the compiler to deal with spreading these apart and determining what could take place in between. The memory access itself was performed by processor instructions local to the memory module, so a reference could be multi-phase with protocol processing (for synchronization or coherence) along the way. A restricted form of this we see today in the context of application specific cache coherence protocols in the Flash and Typhoon work.

Given how the complexity of memory systems is increasing and how processing rates are increasing relative to access times and transfer latencies, it is likely that in the future you will think about memory references the way you think about messages today. They are issued to an external subsystem, carried out asynchronously with some probability of failure, and complete with a well-defined event.

Program the Scheduler

There was one area where Monsoon and ETS was perhaps not radical enough. The underlying

machine had the concept of state-based instruction execution. For example, the behavior of an instruction depended on the state of the frame slot that it referenced. It was straightforward to map dataflow graphs into deterministic instruction sequences. What we did not explore was the power offered by allowing the compiler to generate instruction sequences that were nondeterministic, in that the behavior depended strongly on the order in which events occur within the machine. We explored this idea a little in the Multithreading workshop at Supercomputing 91. Imagine in a conventional instruction set architecture that the register reservation bits are exposed in the instruction set, so it would be possible to branch on the result of a load being 'not yet present'. Or, you might have the ability to branch on a cache miss. In TAM we had the idea that an executing thread could adjust the scheduling of threads by modifying the scheduling data structure. As memory hierarchies become more complex, as lock-up free operation becomes more common, threading becomes more widely used, compiler-based prefetching more common and as adaptive programming techniques become better developed, we may well find that programs behave more like control systems, adjusting the load they place on the machine in response to observed temporal behavior of the machine.

The Future?

The authors remain passionate about deep integration of threading, cheap synchronization, and split-phase memory transactions. It may well be that the continued divergence of processor and memory latencies and the increasing intensity of threads and automatic storage management will lead to a "rediscovery" of the fine-grained, explicitly managed frame model of Monsoon.

Where did the people go?

At the time this paper was published, Greg and David had recently finished their PhDs at MIT. Greg stayed with the dataflow project at MIT, first as a research staff member and then as an Assistant Professor. He later went to Thinking Machines Corporation as Senior Architect designing the follow-on to the CM-5. He is now Chief Technology Officer for Sun Microsystems, Inc.

David became an Assistant Professor at UC Berkeley, where he did TAM, Active Messages, Split-C, LogP, and NOW while working through the academic ranks. He is now a Professor and Vice Chair of Computing and Networking and consults for the CTO of Sun Microsystems.

Acknowledgments

The Monsoon and ETS work would not have been possible without strong federal and industrial support; the Defense Advanced Research Project Agency and Motorola made bold investments in a truly experimental architecture. The National Science Foundation made possible further investigation through PYI and PFF support. The project all started with Arvind's vision of a better basis for computer architecture, which attracted an incredibly talented group of young individuals. We especially want to thank Andy Boughton, Ken Traub, Steve Heller, Vinod Kathail, Keshav Pingali, Richard Soley, and Jamie Hicks.

Related References

[1] G. Papadopoulos and K. Traub, "Multithreading: A Revisionist View of Dataflow Architecture," *Proc. 18th Annual Symposium on Computer Architecture*, pp. 342-351, May 1991.

[2] D. Culler, A. Sah, K. Schauser, T. von Eicken and J. Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. of the 4th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 164-75, Oct. 1991.

[3] R. Nikhil, G. Papadopoulos and Arvind, "*T: A Multithreaded Massively Parallel Architecture.," *Proc. 19th Annual Symposium on Computer Architecture*, pp. 156-167, May 1992.

[4] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Annual Symposium on Computer Architecture*, pp. 256-266, May 1992.

[5] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich and W. Lee, "The M-Machine Multicomputer," *Proc. of the 28th Annual International Symposium on Microarchitecture*, pp. 146-56, Nov. 1995.

[6] G. Sohi, S. Breach and T. Vijaykumar, "Multiscalar Processors," *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pp. 414-25, Jun. 1995.

[7] D. Tullsen, S. Eggers and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pp. 392-403, Jun 1995.

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 207-216, Aug. 1995.

[9] G. Alverson, P. Briggs, S. Coatney, S. Kahan and R. Korry, "Tera Hardware-Software Cooperation," *Proc. of the 1997 ACM/IEEE SC97 Conf.*, Nov. 1997.