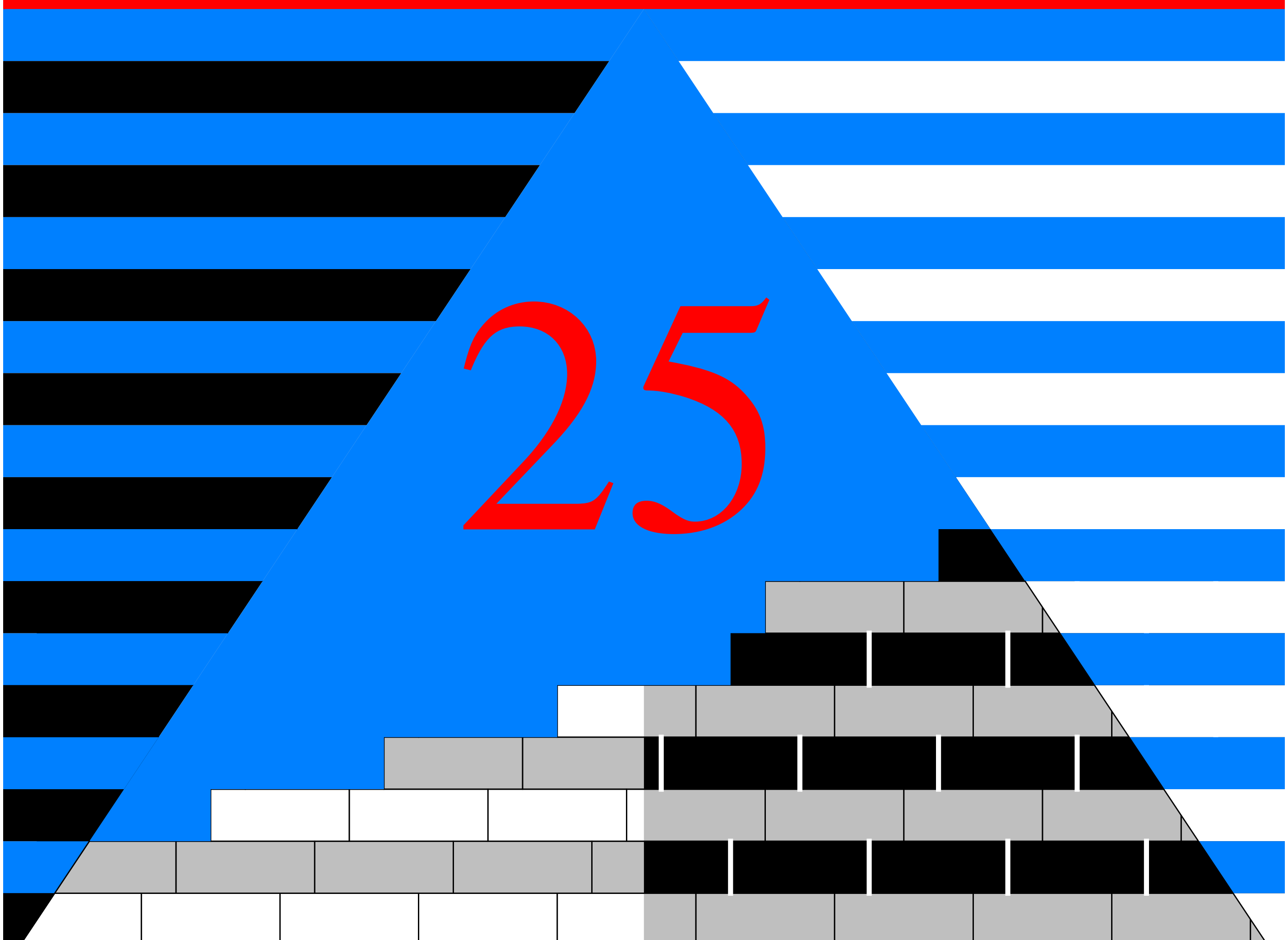
25 Years of the
International Symposia on

COMPUTER ARCHITECTURE

Selected Papers



edited by Gurindar Sohi

Foreword

It has indeed been an honor to put together this volume of selected papers from the International Symposia on Computer Architecture (ISCA). 1998 marks the 25th anniversary of the symposium. During this time the symposium has been the forum for presentation of a significant number of research ideas and results that have had a significant impact on how computing machines are built. This volume is an attempt to bring forth a selection of such papers, along with retrospectives written by the authors of the papers describing, among other things, the history and background of the research reported in the paper.

The papers in this volume were selected by a committee of program chairs/vice-chairs of recent ISCA conferences: Doug DeGroot and Yale Patt (ISCA-15), Arvind (ISCA-16), Jim Goodman (ISCA-17), Allan Gottlieb (ISCA-19), John Hennessy (ISCA-20), Jim Smith (ISCA-22), Norm Jouppi (ISCA-23), and Trevor Mudge (ISCA-24); Janak Patel (ISCA-21) participated for part of the process. We started the process with a list of about 140 papers that were nominated by members of this committee. This list was pruned to about 70 papers with one round of voting, and to the final 41 with further rounds of voting. (The main selection criterion was the impact of the paper. Papers in ISCA-23 and ISCA-24 were considered to be too recent by most committee members, and therefore this collection does not include papers from these symposia.)

This volume would not have been possible without a considerable effort on the part of several individuals, and I would like to thank them for their efforts. The authors of the retrospectives put in a lot of effort into their writeups; without them this volume would not have been possible. Avinash Sodani and Doug Burger did most of the formatting (with help from Jim Goodman) and Eric Rotenberg helped with proofreading. Milo Martin and Harit Modi did the cover design. (The cover pictures a partially-built pyramid with 41 building blocks, corresponding to the papers in this volume, and lots of blue sky for future ISCA papers.) Finally, John Hennessy and Mateo Valero not only shared my enthusiasm for this project but also provided constant encouragement.

List of Selected Papers

ISCA 1 (1974)

ISCA 2 (1975):

ISCA 3 (1976):

ISCA 4 (1977):

ISCA 7 (1980):

ISCA 8 (1981):

9 *Lockup-Free Instruction Fetch/Prefetch Cache Organization*
David Kroft

10 *A Study of Branch Prediction Strategies*
James E. Smith

11 *RISC I: A Reduced Instruction Set VLSI Computer*
David A. Patterson and Carlo H. Séquin

ISCA 9 (1982):

12 *Decoupled Access/Execute Computer Architecture*
James E. Smith

13 *The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine*
Allan Gottlieb, Ralph Grisham, Clyde P. Kruskal,
Kevin P. McAuliffe, Larry Rudolph and Marc Snir

ISCA 10 (1983):

14 *Using Cache Memory to Reduce Processor-Memory Traffic*
James R. Goodman

15 *Very Long Instruction Word Architectures and the ELI-512*
Joseph A. Fisher

ISCA 11 (1984):

16 *A Characterization of Processor Performance in the VAX-11/780*
Joel S. Emer and Douglas W. Clark

17 *A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories*
Mark S. Papamarcos and Janak H. Patel

ISCA 12 (1985):

18 *Implementation of Precise Interrupts in Pipelined Processors*
James E. Smith and Andrew R. Pleszkun

ISCA 13 (1986):

19 *HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality*
Wen-mei Hwu and Yale N. Patt

20 *Warp Architecture and Implementation*
Marco Annaratone, Emmanuel Arnould, Thomas Gross, H.T. Kung, Monica S. Lam, Onat Menzilcioglu, Ken Sarocky and Jon A. Webb

21 *Memory Access Buffering in Multiprocessors*
Michel Dubois, Christoph Scheurich and Fayé Briggs

ISCA 14 (1987):

22 *Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors*
Gurindar S. Sohi and Sriram Vajapeyam

23 *Architecture of a Message-Driven Processor*
William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty and Scott Wills

ISCA 15 (1988):

24 *On the Inclusion Properties for Multi-Level Cache Hierarchies*
Jean-Loup Baer and Wen-Hann Wang

25 *An Evaluation of Directory Schemes for Cache Coherence*
Anant Agarwal, Richard Simoni, John Hennessy and Mark Horowitz

ISCA 17 (1990):

26 *Weak Ordering—A New Definition*
Sarita V. Adve and Mark D. Hill

27 *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*
Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta and John Hennessy

28 *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*
Norman P. Jouppi

29 *Monsoon: An Explicit Token-Store Architecture*
Gregory M. Papadopoulos and David E. Culler

ISCA 18 (1991):

30 *IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors*
Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter and Wen-mei W. Hwu

ISCA 19 (1992):

31 *The DASH Prototype: Implementation and Performance*
Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy

32 *Active Messages: A Mechanism for Integrated Communication and Computation*
Thorsten von Eicken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser

33 *The Turn Model for Adaptive Routing*
Christopher J. Glass and Lionel M. Ni

34 *Alternative Implementations of Two-Level Adaptive Branch Prediction*
Tse-Yu Yeh and Yale N. Patt

ISCA 20 (1993):

35 *The Cedar System and an Initial Performance Study*
D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. -Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews and S. Turner

ISCA 21 (1994):

36 *Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer*
Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten and Jonathan Sandberg

37 *The Stanford FLASH Multiprocessor*
Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum and John Hennessy

38 *Tempest and Typhoon: User-Level Shared Memory*
Steven K. Reinhardt, James R. Larus and David A. Wood

ISCA 22 (1995):

39 *The MIT Alewife Machine: Architecture and Performance*
Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie and Donald Yeung

40 *Multiscalar Processors*
Gurindar S. Sohi, Scott E. Breach and T.N. Vijaykumar

41 *Simultaneous Multithreading: Maximizing On-Chip Parallelism*
Dean M. Tullsen, Susan J. Eggers and Henry M. Levy

Table of Retrospectives

Retrospective:

Banyan Networks for Partitioning Multiprocessor Systems

Jack Lipovski

Electrical and Computer Engineering
University of Texas at Austin
lipovski@ece.utexas.edu

This paper consolidated apparently different areas of research and application involving multistage interconnection networks. It showed that the previously implemented barrel shifter (Illiac IV) and flip network (Staran) were special cases of a general case of interconnection network, and that the general case covered a wide range of other interconnection networks. The class of graphs describing these networks has been studied in Graph Theory, and several practical interconnection networks in this family have been built. Nevertheless, considerable work is yet to be done in both the theoretical and the practical exploration of "Banyan Graphs" and "Banyan Interconnection Networks." I remain optimistic that further research into both the theoretical and practical aspects of this research will someday lead to superior parallel computers.

Retrospective:

A Preliminary Architecture for a Basic Data Flow Processor

Jack B. Dennis

Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge, MA 02139
dennis@aj.lcs.mit.edu

In 1964 MIT's Project MAC selected the General Electric Model 635 computer system as the basis for building the Multics time sharing utility computer system [7]. I had participated in the work of the four-person committee that visited computer manufacturers to evaluate the potential of each for achieving the goals of Multics. The basis for this evaluation was the report of the "Long Range Computer Study Group" completed in 1961 [21], to which I contributed.

Part of my contribution to this effort was my work on segmentation [11] which was motivated by the issues of making computer systems more compatible with the needs of programming in a human interactive environment: the ability to write programs without attention to the details of storage management; and the ability to dynamically share use of system resources. Our experience with CTSS [6] and the PDP-1 time-sharing system [10] had brought these issues to the fore. We were also concerned with parallelism in computing, initially because we had argued that multiple processors could make more effective use of the large memory systems we saw the need for. We insisted that Multics be a multiprocessor system, but the primary argument eventually became one of fault tolerance, the capability of the system to survive the loss of any single module.

In 1964, I formed the "Computation Structures Group" within Project MAC. From the beginning we adopted the principle that the programming interface supported by the hardware and operating system should be one that matches the semantics of the programming language used to write applications [17]. We insisted that the user programming language be expanded to incorporate facilities for expressing concurrency, file handling, and input/output so that the programmer need never go outside the application language to write complete programs of any size. Moreover, we aimed to do this in a fashion that would be consistent with the requirements of modular programming. (We were influenced by the symposium on Modular Programming organized by Larry Constantine in Cambridge, MA in 1968 [5].)

A major benefit at this time was the attitude of DARPA, the sponsor of Project MAC and the Multics development, toward graduate education: the "umbrella" grant under which Project MAC operated allowed faculty and graduate students to pursue ideas independently without hassles with proposals and sponsor reports (other than the Project MAC Annual Report).

One area that attracted my attention was the theoretical area of program schemata. The work of the Russian scientist Ianov was brought to my attention by Joseph Rutledge of IBM [24]. The work of Michael Patterson, at MIT at that time, was also influential. The outcome of this was the formalism of Data Flow Program Schemes written up together with my graduate students, John Linderman and John Fosseen in 1972 [15]. Dataflow schemes, as a model of programming, had many of the qualities we desired for expressing parallelism with modularity. However, they lacked models for the important concepts of functions and data structures. (The addition of data structures to the program schema model was explored in my 1968 IFIP paper [12], and an essentially complete semantic model based on dataflow concepts was published in 1975 [13].)

We learned of the work of Karp and Miller at the IBM Watson Laboratory [19], and of Duane Adams at Stanford [1] later and were delighted and encouraged to find that other computer scientists were working with similar ideas. However, I felt that our model was better aligned to the requirements of modular software. In addition, it was consistent with Edsger Dijkstra's principles of structured programming [8], and could be used as a semantic model for functional programming [4].

About this time Professor Barry Vercoe of the MIT Humanities Department was actively involved in the art of computer-aided music sound synthesis, and was frustrated by the lack of sufficient performance from available computers to create true symphonic sound using computer programs. I took this as a challenge and used this application as an attractive area in which to try out our concepts.

Our first concept of a dataflow architecture was based on a very simple class of Dataflow Schemata: those that would operate continuously without decisions. (Al Davis built an interesting early data flow machine at the University of Utah [9].) A paper on the implementation of this simple architecture would win a best paper prize for my coauthor, David Misunas [20]. The paper for the 1975 ISCA arose from our efforts at extending our concepts to overcome limitations of earlier architectures. As we discussed in *Project MAC Progress Report XI* [22], we had in mind a succession of extensions to encompass the full semantic model we had developed. In particular the 1975 paper incorporated mechanisms for handling loops and conditionals, and the thesis of James Rumbaugh [25] detailed a proposed architecture that would implement the complete semantic model.

Soon after publication of the ISCA paper, we realized that building enough instruction cells to accommodate really large programs would be impractical for a general purpose processor. As a result several variations were explored. The event that spread interest in the ideas, however, was the 1975 Sagamore Conference where I held a long extemporaneous seminar on dataflow ideas. This, I believe, led directly to the dataflow projects in Manchester, England, in Japan, and to the beginning of Arvind's work at the University of California at Irvine [3].

We also soon realized that scheduling each dataflow instruction independently was too inefficient, not because of the scheduling overhead, however, but because we were not able to exploit the advantage of transferring values between instructions through processor registers. This problem can be addressed by some means for grouping instructions together, as discussed in [14], and advocated by Robert Iannucci [18]. The ideas of fine grain multithreading [16], as introduced in the Tera computer [2], are the legacy of this idea, and Arvind's group at MIT implemented a prototype in the Monsoon system [23]. Nevertheless, the full advantages of having a complete semantic model consistent with the requirements of modular programming have yet to be achieved. My optimism leads me to believe that the next decade will finally see these ideas reach fruition.

A key ingredient in a satisfactory semantic model for general purpose computing is the ability to share any kind of memory object among any number of independent applications. It is only gradually becoming appreciated that this calls for systems that implement a global address space. At present there exists a single commercial example of such an architecture: the IBM AS/400 [26], and interest in this aspect of computer architecture in university research is at a nadir. The AS/400 is inefficient and does not address the issues of software modularity raised by parallel processing, and is inadequate in its support of dynamic memory management. There remain many significant opportunities in computer architecture. The excitement continues!

References

[1] Duane A. Adams. A Computation Model with Data-Sequenced Control. Technical Report CGTM 45, Stanford University, May 1968.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, IEEE, 1990, pages 1-6.

[3] Arvind, K. Gostelow and W. Plouffe. *The (preliminary) Id report: An asynchronous programming language and computing machine*. Technical Report 114, Department of Information and Computer Science, University of California, Irvine, September 1978.

[4] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8:613-641. August 1978.

[5] Larry Constantine, Editor. *Modular Programming: Proceedings of a National Symposium*. Cambridge, MA: Information and Systems Press, 1968.

[6] Fernando J. Corbato, et al. *The Compatible Time-Sharing System: A Programmer's Guide*. Cambridge, MA: M.I.T. Press, 1963.

[7] Fernando J. Corbato and Jerome H. Saltzer. Multics: The first seven years. In *AFIPS Conference Proceedings 40: Spring Joint Computer Conference*, 1972. pages 571-583.

[8] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. New York: Academic Press, 1972.

[9] Al Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the Fifth Annual Symposium on Computer Architecture*. IEEE, 1978, pages 210-215.

[10] Jack B. Dennis. A multi-user computer facility for education and research. *Communications of the ACM 7*, 9 (September 1964).

[11] Jack B. Dennis. Segmentation and the design of multi-programmed computer systems. *Journal of the ACM 12*, 4 (October 1965).

[12] Jack B. Dennis. Programming generality, parallelism, and computer architecture. In *Information Processing 68*. Amsterdam: North-Holland, 1969, pages 484-492.

[13] Jack B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science, Volume 19: Programming Symposium*. B. Robinet, Ed. Berlin: Springer-Verlag, 1974, pages 362-376.

[14] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2. Prentice-Hall, 1991.

[15] Jack B. Dennis, John Fosseen, and John P. Linderman. Data flow schemas. In *International Symposium on Theoretical Programming*. *Lecture Notes in Computer Science*, No. 5. Berlin: Springer-Verlag, 1972, pages 187-216.

[16] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Architecture*. Kluwer, 1994.

[17] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multi-programmed computations. *Communications of the ACM 9*, 2 (February 1966).

[18] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE Computer Society, 1988.

[19] Richard M. Karp and Raymond Miller. Properties of a model for parallel computation: determinacy, termination, queueing. *SIAM Journal of Applied Mathematics 14*, 6:1390--1411, November 1966.

[20] David P. Misunas. Petri nets and speed independent design. *Communications of the ACM 16*, 8:474-481, August 1973.

[21] M.I.T Computation Center. Report of the Long Range Computer Study Group. April, 1961.

[22] M.I.T Laboratory for computer Science. Progress Report XI. 1974, pages 84-90.

[23] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 1990, pages 82-91.

[24] Joseph D. Rutledge. On Ianov's program schemata. *Journal of the ACM* 11, 1:1-9, January 1964.

[25] James Rumbaugh. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Technical Report MIT/LCS/TR-150, M.I.T. Laboratory for Computer Science, Cambridge, MA, May 1975.

[26] Frank Soltis. *Inside the AS/400*. Loveland, CO: Duke Communications, 1996.

Retrospective:

Improving the Throughput of a Pipeline by Insertion of Delays

Janak H. Patel

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801
patel@crhc.uiuc.edu

Origins of this Paper

Present times are the best of times for the computer and electronics industry. It was not always that way. In 1971 the computer and electronics industry was in one of its worst years. As a young graduate from Stanford University with a fresh M.S. in Electrical Engineering, I could not find a suitable job. So I decided to go back to Stanford to continue graduate studies beyond a Masters, still not sure if I wanted to do a Ph.D. I talked to one of the Stanford professors for doing some project for the Engineer degree which is a terminal degree for doing more than Masters but not doing thesis research. The professor I talked to was Edward S. Davidson. I told him that I am just an engineer, I don't want to do Ph.D. research. In reality I had no idea what Ph.D. research involved since my M.S. degree did not require a thesis. In that summer of 1971, Davidson gave me some papers on pipelining. In about a month or two, I wrote down some simple ideas extending the elegant reservation table and scheduling method that Davidson had devised. I handed him the hand written paper explaining how one can modify the reservation tables to get a desired schedule. This is when he asked me, "do you know what research is?" I was still not sure what it was, and I told him so. In fact, I told him this is why I was doing an Engineer degree, and not a Ph.D. degree. That is when he pointed to my papers and told that what I was doing was in fact research! He told me if I continued along the same path, I could have a Ph.D. thesis. I was finally convinced. And so I had a Ph.D. thesis topic and an advisor. Now that I am a Professor myself, I do occasionally use this method to convince a reluctant young engineer to pursue Ph.D.

Job Shop and Machine Shop Scheduling

This paper is one part of my Ph.D. thesis at Stanford. It builds on the work of Davidson on pipeline scheduling. My goal was to maximize the throughput by modifying the schedule and/or the pipeline. Most of the research work in this area is either in Machine Shop/Job Shop Scheduling and Processor Scheduling. After all a pipeline is like an assembly line. However, a job shop is not like an assembly line. The big difference is, job shop schedules a very small number of jobs each of which execute only once, while an assembly line has thousands of jobs and they are repetitive. To my surprise, the research literature was all about job shop and almost none on assembly line scheduling! And so I had to do this research from scratch without any extensive literature. In fact the end result is that this research could be useful to modern assembly lines! Even today I get an occasional enquiry from a Business school or Industrial Engineering colleague asking me if they could apply this work to industrial production lines. I am not sure if any of them have found a direct application of this work, however, I am very sure that this work has found application in all sorts of computer architecture problems that involve some form of resource sharing and scheduling.

Modern Applications of this Work

If one is not put off by the math of the paper, which only involves very basic number theory, one will find that it is quite easy to understand and use the results in a variety of applications which involve resource sharing among several tasks. Several researchers have used the techniques of this paper in their work. Bab Rau and Dave Yen applied this to scheduling in the PolyCyclic architecture CYDRA. Peter Kogge has applied it to microcode optimization. Bab Rau, Wen-Mei Hwu and Ed Davidson have found its use in Software Pipelining. Vijay Madisetti has found it useful in Digital Signal Processor design. Others have found its use in high level synthesis.

Retrospective:

What Have We Learned from the PDP-11 —
What We Have Learned from VAX and Alpha

Gorden Bell

Senior Researcher
Microsoft Corp., Bay Area Research Center
San Francisco, CA

W.D. Strecker

Sr. VP and Chief Technology Officer
Digital Equipment Corporation,
Maynard, MA

Introduction

The PDP-11, VAX-11 (usually just VAX), and Alpha have been the strategic computer hardware architectures of Digital Equipment Corporation (DEC) from the early 1970's to the present. Although it would be a stretch to consider them variants of a single computer architecture, there are enough common properties in the architectures themselves and in the major software systems supporting the architectures to consider them members of an architecture family.

Our paper "Computer Structures: What Have We Learned from the PDP-11" [1] was written at the time the VAX architecture was being developed, and the learning reported in that paper would strongly influence the design of the VAX architecture.

In this retrospective, we will review how the PDP-11 learning influenced VAX. Next we will discuss what we learned from the VAX architecture. Finally we will discuss the design of the Alpha architecture and how its design and entry into the market resulted not only from VAX learning, but also from environmental factors inside DEC.

PDP-11

The PDP-11 is a CISC architecture with a 16-bit virtual address. The first PDP-11 implementation – the PDP-11/20 – was introduced in 1969. In [2] there is a detailed discussion of the goals and constraints for the design of the PDP-11. Deliberately oversimplifying, these include: (1) provide the ability to build processors with a wide range of performance and function, (2) provide efficient (8-bit) byte processing, (3) provide a flexible, compiler friendly programming model, and (4) provide a flexible I/O structure.

The PDP-11 architecture is a general register design (8 16-bit registers) with the program counter and stack pointer located in the general registers. An elegant set of register-based memory addressing modes combined with the general register structure to produce an architecture that can be programmed as a stack machine, a general register machine, or a memory-to-memory machine. Memory is addressable to the (8-bit) byte and the conditional branch mechanism is based on condition codes. I/O is handled by providing I/O device registers with memory addresses: the registers can then be manipulated by ordinary instructions.

The PDP-11 was a major commercial success, providing the majority of DEC's growth, revenues and profits from the early 1970's to the early 1980's. Also, the PDP-11 significantly influenced computer architecture with its elegant addressing modes and its I/O structure.

The PDP-11 architecture proved to have two real limitations. The first was the 16-bit virtual address space. This will be discussed in the next section. The second was the instruction set and the instruction set encoding. The original PDP-11 had operations to move, add, subtract, compare, and conditional branch on 8- and 16-bit integers. These operations together with the addressing modes were encoded in such a way to effectively exhaust the code space of the PDP-11 instruction format.

This situation made it impossible to compatibly extend the PDP-11 with any consistency or efficiency. The addressing modes could not reasonably be extended or redefined to support a greater than 16-bit virtual address. It was impossible to efficiently add additional instructions in a manner architecturally consistent with the basic instruction set. When certain additional instructions and other capabilities were needed to meet market requirements (e.g. extended integer arithmetic, floating point, and memory management) they were added as implementation specific options and often weren't compatible across implementations. The result of all this was that the PDP-11 was not compiler friendly (given the state of DEC and industry compiler technology in the 1970's). Most PDP-11 language processors were either (1) interpreters or (2) compilers that compiled to an intermediate form that was interpreted at run time.

Far surpassing the lack of architectural control and consistency in PDP-11 hardware implementations was the state of PDP-11 software. Depending on how one counts, there were about 4 operating system families with about 10 named variants. These operating systems supported an arbitrary variety of sometimes incompatible language processor, data management, and transaction processing software. It was understood that this situation would be completely unmanageable and could not be afforded for the future.

From the PDP-11 to VAX

The Bell and Strecker paper [1] has often been quoted because of this statement:

"*There is only one mistake that can be made in a computer design that is difficult to recover from – not providing enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. Of course, there is a fundamental rule of computer (and perhaps other) designs that helps to alleviate this problem: any well-designed machine can be evolved through at least one major change. It is extremely embarrassing that the PDP-11 had to be evolved with memory management only two years after the paper was written outlining the goal of providing increased address space. All predecessor DEC designs have suffered the same problem and only the PDP-10 evolved over a ten year period before a change was made to increase its address space*."

By 1975, the PDP-11's 16-bit virtual address had become a real limit for applications. (Various approaches had been used in the PDP-11 implementations to extend the physical address space to beyond 2^{16} bytes, but they did not solve the application problem.)

Moore's Law [3] predicts that DRAM chip capacity increases 4 times every 3 years. Thus, if memory chip prices are constant, and if users pay a constant amount for computers, then the number of address bits needed to address a constant price memory will grow by one address bit every 18 months.

If a 16-bit address was reaching its limit in 1975, then one could determine the likely lifetime of any address size expansion. We defined the PDP-11's successor — VAX — to have a virtual address of 32 bits. Thus we concluded that the VAX architecture – based upon the above model that the only fundamental limitation on architecture lifetime is addressing – should comfortably last about 24 years: until 1999. It would turn out that the limit on the VAX architecture lifetime wasn't the size of the virtual address.

VAX

The VAX is a CISC architecture with a 32-bit virtual address. The first VAX implementation – the VAX-11/780 – was introduced in 1978 [4]. The design of VAX was started in 1975. The overarching goal was to produce a 'compatible' extension of the PDP-11 that would solve the virtual address space limitation of the PDP-11 (the name VAX-11 is derived from 'Virtual Address eXtension of the PDP-11). The principal constraint on the design of VAX was that – despite the doubled virtual address size – VAX code would be no bigger than equivalent PDP-11 code.

A strong goal was to eliminate the chaos of the PDP-11 software. This was approached in three ways. (1) VAX was to have a single strategic operating system — VMS — with real-time, time-sharing, and transaction processing capabilities. (2) To make the VAX compiler friendly (again in the context of DEC's mid-1970's compiler technology), an extreme focus was placed on instruction set completeness and regularity. (3) The VAX software environment was to be based on the model that any software can 'call' any other software. To strongly motivate software developers to follow this model, VAX defined a number of 'software' data types (i.e. subroutine stack frames, queues, variable length bit fields, character strings, and decimal strings) and provided instruction support for these data types. 'Software' is used in the sense that most applications would see little performance degradation if the data types were implemented in software.

Given the code size constraint and the limitations discussed above on extending the PDP-11 instruction set, the VAX instruction format is not a superset of the PDP-11 instruction format. Instead a new instruction format was designed for VAX and formal PDP-11 compatibility was provided by a tightly integrated PDP-11 compatibility mode that allowed execution of PDP-11 instructions in the VAX virtual address space.

The VAX architecture has the same general structure as the PDP-11 – general registers (extended to 16 32-bit registers), with the program counter and stack pointer located in the register set, a rich set of register-based addressing modes (extended to include scaled indexing and an efficient encoding of literals). VAX also has the same data types, condition codes, and byte addressing as the PDP-11.

As introduced above, VAX extended the PDP-11 by defining new data types – queues, variable length bit fields, subroutine stack frames, character strings, and decimal strings – and a complete set of instructions to operate on these data types.

To achieve the code size constraint, VAX defined an extremely space efficient method of encoding instructions. Instructions are provided in multiple forms with implicit and explicit operands. For example, the 32-bit integer ADD instruction is provided in the following forms:

increment	A	; add 1 to A
add	A, B	; add A to B
add	A, B, C	; add A to B and store the sum in C

Every explicit instruction operand (A, B, and C in the previous example) is specified in a general way using any of the VAX addressing modes. This leads to the VAX instruction format of

[*opcode, operand-1-specifier, …, operand-n-specifier*]

where *n* is the number of explicit operands. VAX instructions were defined with 0-6 explicit operands.

VAX was a huge commercial success. VAX provided the majority of DEC's growth, revenues and profits from the early 1980's to the early 1990's. The success of VAX was clearly inseparable from the VMS operating system. VMS was a true 32-bit virtual memory operating system that performed well from its first release. VMS embedded DECnet such that it was transparent to applications whether any file or other I/O operation was local or over the network. On top of VMS was a highly integrated (and network-transparent) software set including multiple language compilers, a database (RDB), transaction processing (ACMS), and an 'integrated office' environment (ALL-IN-1). VMS invented and first implemented the now-pervasive concept of clusters [5,6].

When it was clear that the market was responding to VAX and VMS, DEC moved to VAX and VMS as its sole computer system strategy [7]. All other DEC systems were put in niche roles or moved to what was essentially a maintenance mode.

It is notable that the VAX architecture remained essentially unchanged over the last 20 years. The only material addition was the early adding of 2 floating-point data types. The only other material change was to define permissible subsets [8] of the architecture. (The MicroVAX architecture was such a subset.) Processors implementing VAX subsets generate sufficient state on encountering an instruction not in the subset to enable transparent, efficient software interpretation of the missing instruction.

A retrospective on VAX must reflect a strong sense of time. The VAX embodiment of the goals and constraints of PDP-11 compatibility, the code size constraint, instruction set completeness and regularity, and hardware support for 'software' data types was absolutely key to moving from the PDP-11 (and other DEC architectures) to the hugely successful VAX and VMS business. However, once that success was achieved, the VAX architecture carried a complexity burden that would make it particularly vulnerable to the RISC concept.

From VAX to Alpha

In 1980, Patterson [9] discussed the modern RISC architecture concept. By the mid-1980's, there was a general consensus in DEC that for a given amount of CPU logic in a given technology, a RISC processor could achieve (at least) twice the performance of a CISC processor. There was no consensus, however, on what to do about this.

There were two rational strategic responses to the RISC challenge:

1. Since the RISC advantage would 'only' be a factor of 2, DEC could 'tough it out' until the limits of a 32-bit address space would force a new architecture (predicted, as discussed above, to be about 1999). To 'tough it out' would be (1) to focus on the most aggressive possible microprocessor implementations of VAX, (2) to use multiprocessing and clustering to achieve performance, and (3) to accept limited success in some market segments where (1) and (2) would be marginal (e.g. UNIX workstations). Effectively, this was the type of strategy successfully employed by IBM for the '360' architecture and Intel for the 'x86' architecture – the other two important CISC architectures with a large customer base.

2. Since CISC was going to 'lose' by at least a factor of 2 to RISC, it was essential to embrace RISC ASAP, define a new or use an already defined RISC architecture, and get products to the market in a timely manner. Especially important, it was necessary to get DEC's then strategic software system – VMS – ported to the RISC architecture (or perhaps, even better, made processor independent, and sold industry-wide). Effectively, this was the type of strategy successfully employed by Sun in moving from the '68000' to SPARC.

As we discussed in the last section, DEC's VAX business was a huge success, and it was very profitable. At this time, DEC's senior leadership was operating under the philosophy best captured by the two phrases 'if some strategy is good, less strategy is better,' and 'if some internal competition is good, more internal competition is better.'

As a result, exploiting the considerable profits of the VAX business, an overwhelming array of internal projects in processor technology, VAX and RISC architectures, and operating systems were launched. During the second half of the 1980's, major projects were undertaken in various combinations of 3 different ECL gate array technologies, high performance multichip packaging, advanced custom CMOS, 3 internally developed and one externally developed (MIPS) 32-bit RISC architectures, a 64-bit RISC architecture (Alpha), multiple system and I/O busses, a new UNIX operating system, and two new proprietary operating systems! Knowing that all these projects could not possibly be successful, DEC's product development organization was locked in internecine warfare.

By the end of the 1980's DEC had essentially lost control of its system strategy. It wasn't explainable or affordable, and remarkably still hadn't done everything necessary to successfully implement either of the two strategic alternatives discussed above. It wasn't until 1992/1993 that DEC changed its senior leadership and regained control of its system strategy.

Alpha

Alpha is a RISC architecture with a 64-bit virtual address [10]. The first Alpha implementation – the 21064 single chip microprocessor – was introduced in early 1992. Computer systems using the 21064 were introduced at the end of 1992.

The goals of the Alpha architecture design were high performance, longevity, support for running the VMS and UNIX operating systems, and support for existing VMS and UNIX applications. The goals of high performance and longevity were met by a RISC approach with extreme attention to details that might interfere with high-speed implementations, a 64-bit virtual address, and PALcode (to be discussed later). The goals of VMS/VAX and ULTRIX/MIPS (DEC's UNIX offering was called ULTRIX and ran on the MIPS architecture) application support were met with data type and addressing compatibility with VAX and (little-endian) MIPS, PALcode, and binary translation (discussed later).

Compared to VAX, the design of Alpha can be considered 'classic' RISC. There are 32 64-bit general-purpose registers and 32 64-bit floating point registers. All instructions are 32 bits in length. The programming model is load/store: the only memory operations are load from memory to register and store to memory from register. All other operations are between registers. The only data types are integer and floating point with VAX compatibility. The principal integer data type is 64 bits, and there is very limited instruction support for smaller integers. Memory is addressable to the byte but there are strong size and alignment constraints on memory accesses. There are no condition codes: conditional branches are based on testing the state of a register.

In addition to 'classic' RISC techniques, Alpha has some novel approaches for enabling high-speed implementations. For example, there is a very flexible approach to specifying and handling arithmetic exceptions. A conditional move instruction eliminates branches in certain instruction sequences. Certain instructions contain hints about branch targets and data prefetching.

PALcode (Privileged Architecture Library) provides a means of implementing the privileged architecture seen by an operating systems. Privileged architecture includes context switching, interrupts, exceptions, and memory management. In Alpha, PALcode is implemented with ordinary instructions running in physical memory, with interrupts off, and access to all machine state. The PALcode is tailored to the needs of each operating system (e.g. VMS, UNIX, and Windows NT).

Rather than hardware compatibility modes, binary translation is used to run VMS/VAX-based and ULTRIX/MIPS-based applications on Alpha. The binary translator takes, say, VMS/VAX-based executable code and compiles it to the extent possible to VMS/Alpha-based executable code. A runtime interpreter paired with an incremental compiler handles the portion of the code that cannot be initially compiled. During runtime interpretation, enough additional information and context is gathered to significantly extend the scope and optimization of the initial compilation.

Binary translation was very successful in executing applications from VAX and MIPS on Alpha. Recently it has been used successfully to execute Windows NT/x86 applications on Windows NT/Alpha.

Around Alpha a unified system strategy was developed. The strategy consisted of (1) an aggressive long-term road map for Alpha microprocessors, (2) a family of systems from workstations to large-scale multiprocessor systems using the Alpha microprocessor. (3) PCI bus-based I/O for all systems, and support by three operating systems: VMS (evolved from 32-bit to 64-bit support), UNIX (called DIGITAL UNIX: a new pure 64-bit operating system based on OSF technology) and Windows NT (provided by Microsoft). All other DEC systems were put in niche roles or moved to what was essentially a maintenance mode.

This strategy was well executed by DEC. The various Alpha microprocessors maintained a significant performance lead over competitive RISC and CISC microprocessors. The transition of VMS/VAX to VMS/Alpha and ULTRIX/MIPS to DIGITAL UNIX/Alpha went smoothly. In application areas – particularly databases - where 64-bit addressing could be exploited, Alpha performance and 64-bit DIGITAL UNIX functionality set the competitive benchmark.

Unfortunately, the strategy was late. By 1992/1993 *de facto* standards had been set by competitors for RISC and (32-bit) UNIX. Despite the simplicity of the strategy, and the technical excellence, DEC would struggle to get product volumes adequate for a profitable systems business.

However, a new opportunity is emerging for Alpha. As the UNIX and the Windows NT market moves from 32 to 64 bits, Alpha is the only mature, high performance 64-bit RISC architecture with pure 64-bit UNIX and Windows NT support. A complete retrospective on Alpha awaits the industry 32- to 64-bit transition.

Summary

The PDP-11, VAX-11, and Alpha can be considered members of an architecture family starting in the 1960's and extending to the present.

The PDP-11 was a huge commercial success for DEC. The PDP-11 was the *de facto* standard 16-bit minicomputer in the 1970's. The basic PDP-11's design was extremely elegant and it significantly influenced future computer architecture. However, the PDP-11's 16-bit virtual address space and the inability to efficiently and consistently extend the architecture, led to its successor – VAX – being designed only 6 years after its introduction.

The VAX was similarly a huge commercial success for DEC. VAX and its closely related software system – VMS – became the *de facto* standard for 32-bit virtual memory networked computing in the 1980's. However, VAX, driven by its initial design goals and constraints, was a complex architecture, and was particularly challenged (internally and externally) by the RISC concept that competitively emerged in the mid-1980's.

DEC's internal situation in the second half of the 1980's made it impossible to achieve a timely, rational response to the RISC challenge. By the time the Alpha strategy emerged in 1992/1993, DEC had lost momentum in the market and other vendors had established *de facto* standards in RISC and UNIX. This situation would impact the commercial success of Alpha despite its superior technical attributes. However, the Alpha story awaits completion of the industry transition from 32 to 64 bits, starting – as we predicted in 1975 – in about 1999.

References

[1] G. Bell W. D. and Strecker, "Computer Structures: What Have We Learned from the PDP-11," *The 3rd Annual Symposium on Computer Architecture Conference Proceedings*, pp. 1-14, 1976.

[2] C. G. Bell, R. Cady, H. McFarland, B. Delagi, J. O'Laughlin, R. Noonan and W. Wulf, "A New Architecture for Mini-Computers - The DEC PDP-11," *Proceedings of the Sprint Joint Computer Conference*, pp. 657-675, AFIPS Press, 1970.

[3] R. R. Schaller, "Moore's Law: Past, Present, and Future," *IEEE Spectrum*, pp.52-59, June 1997.

[4] W. D. Strecker, "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proceedings of the National Computer Conference*, pp. 967-980, AFIPS Press, 1978.

[5] W. D. Strecker, "Clustering VAX Superminicomputers Into Large Multiprocessor Systems," *Electronics*, pp. 143-146, October 20, 1983.

[6] N. Kronenberg, H. Levy, W. Strecker, "VAX Clusters: A Closely Coupled Distributed System," *ACM Transactions on Computer Systems,* May 1986.

[7] C. G. Bell, "Toward a History of (Personal) Workstations," *ACM Conference on the History of Personal Workstations*, January 9, 1986, published in Goldberg, A., A History of Personal Workstations, Addison-Wesley, Reading, MA, 1988.

[8] T. E. Leonard, *VAX Architecture Reference Manual*, DEC Books, Burlington, MA, 1987.

[9] D. A. Patterson and D. R. Ditzel, "The Case for the Reduced Instruction Set Computer," pp. 25-33, *Computer Architecture News*, October 1980.

[10] R. L. Sites, *Alpha Architecture Reference Manual*, DEC Press, Burlington, MA, 1992.

Retrospective:

An Instruction Timing Model of CPU Performance

Leonard J. Shustek

Network Associates, Inc.
Portola Valley, CA 94028
Len_Shustek@NAI.com

Bernard L. Peuto

Concord Consulting,
Portola Valley, CA 94028
blpeuto@peuto.com

Reading one's own work from 20+ years ago results inevitably in a mixture of pride and embarrassment, recognition of achievements and realization of opportunities missed. This paper is no exception.

What We Did

There are three main contributions which this paper attempted to make:

1. Measurements of dynamic as well as static instruction utilization for a popular mainframe instruction set, which was then extended in the related Ph.D. thesis [1] to other instruction set architectures including microprocessors. There were few other comparable empirical studies in the literature, and we were pleased that our measurements were later used both for instruction-set design and for instruction implementation optimization.

2. Observations on the design of high-performance instructions sets that were implied by the measurements, which were again expanded upon in the thesis. The overall conclusion, counter to the movement in the 60's and early 70's towards complex instruction sets which reduced the semantic gap between programming languages and the hardware, was that "simpler is better". We like to flatter ourselves in thinking that this study was one of the instigators of the RISC movement.

3. An "instruction timing" model of CPU timing analysis that was much simpler than a full simulation of the implementation but more accurate than just counting instructions. This hybrid model used timing formulas derived from the implementation (whose data- dependent parameters were measured, such as average string lengths), augmented by simulations of subsystems that depended on instruction sequences, such as cache memory interlocks and instruction prefetch. The result was an easy-to-compute prediction of execution time that could be quickly adapted for proposed changes in the implementation.

The model never was used in practice for that last purpose, perhaps because neither of the authors ever again worked for a mainframe manufacturer. Although this timing model was the motivation for the paper's title, in retrospect it was the weakest of the three contributions.

Why We Did It

Published papers often omit motivation which is deductively irrelevant but nevertheless historically interesting. The true progenitor of this work, strangely not referenced in the paper, was microprogrammed instrumentation for instruction set measurements on the Standard Computer Corporation IC-7000 processor, which emulated the IBM 7090 [2]. Analysis similar to what is in the 1977 paper had been done in 1971, for an instruction set which was already obsolete at the time!

The IBM 360/370 instruction-set simulator used to generate the data for this paper had originally been written, after a suggestion by Forest Baskett, to generate address traces for memory system analysis. For years afterwards there was a healthy underground exchange of these address traces. They were used as the basis of scores of papers, including those of Alan Smith on cache design. (Thanks, Alan, for always acknowledging the source of the data — not everyone did!)

A practical motivation for this work was that one of us [BP] was a computer architect at Amdahl Corporation doing performance evaluation of the new 470/V6, an IBM 370 clone. There was little detailed information on instruction-level performance evaluations that was publicly available. Performance measurement at the time either used synthetic benchmarks which were simple but not representative, or a suite of real programs which was difficult to run and yielded no detailed information useful for instruction design tradeoffs. For example, little data was available contrasting the difference between static and dynamic frequencies. At the time, being a computer architect was still synonymous with designing a new computer with a new instruction set, and little else. It was quite exciting to satisfy our architectural curiosity at the instruction level by using customer's benchmarks, and to be able to accurately approximate the execution time for the benchmark.

As with many such projects, the programming was seductive and at least as motivational as the promise of results (or published papers). It was great fun for the programming member of the team [LS]. The paper only hints at the challenges in simulating the least well-specified instruction of the architecture: the operating-system defined Supervisor Call (SVC), which didn't even have the decency to always return to the following instruction. In contrast to the kludgy specialized code to handle SVC was the elegance of a multitasking trace-analysis coroutine system which obviated the need to generate long trace files. Those "engineering" details were, as is traditional, left as an exercise to the reader.

What We Accomplished, and What We Didn't

The paper (and the later thesis) was used by various architects primarily because it had a potpourri of interesting measurements of instruction-set usage by real programs, not because of the "instruction timing model" that made it seem more academic. How long are strings? How far do branches go? In which direction? How many instructions execute between successful branches? How much cache is purged by operating system calls? How often do operands overlap? How many registers are loaded or stored at one time? What are the dominant instruction pairs? These are all basic questions that instruction set designers and implementers needed to know.

The data was used immediately at Zilog in the design of two microprocessors. The Z8000 [3] was an early 16-bit microprocessor, and the dynamic data on memory vs. register operands, instruction time distribution, and branch frequencies helped to tune a design which was still dominated by the need for compatibility with the earlier Z80 and 8080. The Z8 [4] was a single-chip microprocessor that is still in use today. For it, where the freedom to start the design from scratch was tempered by the constraints of a chip with only 128 bytes of RAM, 2K bytes of ROM, limited logic and a slow clock, both the static and dynamic measurements led towards a RISC-like design.

In retrospect we should have recognized more fully the importance of the paper's data, and expanded it into a systematic and comprehensive database of instruction-set measurements. The thesis treated some other instruction set architectures, but we could have made many other measurements, analyzed other applications, included operating-system code, and made both the technology and the results more widely available. We might have standardized measurements for instruction sets the way Whetstone and SPEC metrics standardized program benchmarks.

But one of us (LS) had a doctorate to finish and a paying career to start, one of us (BP) had microprocessors to design, and those goals dictated other directions. We had to be content with what we did, and we are honored that it has been judged suitable for inclusion in this collection.

References

[1] Leonard J. Shustek, *Analysis and Performance of Computer Instruction Sets*, Ph.D. Thesis, Stanford University, January 1978

[2] H. J. Saal and L. J. Shustek, "On Measuring Computer Systems by Microprogramming," in *Microprogramming and Systems Architecture*, pp. 473-490, Infotech Information Ltd, 1975, ISBN 8553-9220-7.

[3] Bernard L. Peuto, "Architecture of a New Microprocessor", *IEEE Computer*, V12, #2, February 1979, pp 10-21.

[4] Charles Bass, Judy Estrin, Bernard L Peuto, and Gary Prosenko, "Introducing the Z8," *Electronics*, August 31, 1978.

Retrospective:

A Retrospective on High-Level Language Computer Architecture

David R. Ditzel

Transmeta Corporation
Santa Clara, CA 95054
dave@transmeta.com

David A. Patterson

Computer Science Division
University of California, Berkeley, CA 94720
pattrsn@CS.Berkeley.EDU

Written at the end of 1979, this paper challenged the 1970's trend in computer architecture toward ever increasing complexity. The decade of the 1970's was dominated by the growing influence of micro-coded minicomputers such as the Digital Equipment Corporation VAX computers. The VAX had over 200 instructions with complex addressing modes. Micro-coded implementations of computer instruction sets meant that adding new instructions was relatively easy.

Research in computer architecture during the 1970's often proposed far more complicated computer hardware that would move towards implementing high level languages directly in hardware in order to facilitate programming in high level languages. Recall that during the 1970's there still existed considerable debate as to whether programming in high level languages would be too inefficient in both performance and code space.

This paper reflected a mood that something was wrong with the direction of computer architecture research. The paper started by repeating six commonly held beliefs ("Axioms") of the day that we felt were not well justified, and ended with an appeal that what mattered was the effect of the combined hardware/software system, and not which individual component was implemented in hardware. In many ways, this retrospective set our direction toward the RISC movement of the 1980's, and heavily influenced us in our research and further publication on RISC processors. Our first paper on RISC was published only five months after this first retrospective paper appeared.

Re-reading this paper 18 years later, we are surprised by how well it holds up. Most surprisingly, many of the same issues about High-level Language Computers are again resurfacing with proposals to implement JAVA byte-codes directly in hardware. Stack machines, byte-coded instruction sets and small code size were the hallmarks of many of the papers calling for High-level Language Computers. We hope that the next generation of computer designers can learn from the lessons of the past, and not repeat the same mistakes over again.

The paper starts with a criticism of the High-Level Language Computer work, including a summary of the High-Level Language Computer motivation as six High-Level Language Computer axioms, followed by our responses to each axiom.

From today's perspective, the major observation about making sure a High-Level Language Computer can execute multiple programming languages is still a good one. It is not clear whether the standard JAVA byte-code instructions are also appropriate for implementing languages like C or C++. This was a common weakness for many of the proposed high level language computers of the 1970's, and we suspect may be a weakness for JAVA processors also.

Another axiom discussed the focus on code size. This issue has not disappeared. In 1998 the processor-memory performance gap is so large that it might make sense to return to very compact instruction encoding just to reduce the number of instruction cache misses. Embedded applications, which were not the focus of that paper, will also desire smaller code to reduce the total system cost. Hence the 16-bit versions of standard 32-bit instruction sets of ARM and MIPS were invented in the last 2 years.

The paper then makes one of our main points, that we should be defining a High-level Language Computer System combined of both hardware and software, and proposes a specific definition.

Our paper proposes metrics to measure difference when a system does or does not obey the definition. For example, imagine the differences in

CPU time, code size, and compile time between compiling with debugging flags on (and no optimization) versus compiling with highest level optimization.

Given 18 years of improvements in processor performance and DRAM capacity with little improvement in software productivity, when people use computers that are 1000 times faster and DRAM with 4000 times the capacity there is less concern about shipping highly optimized code. Hence today we are at a funny point, where people ship either unoptimized or slightly optimized software yet the benchmarks are performed with the highest possible levels of optimization. We can't help but wonder whether these proposed metrics, which didn't particularly catch on, would be just as interesting for machines of 1998 as they were for machines of 1980.

The last part of the paper lists ten attributes that we hoped would be found in High-level Language Computer Systems of the mid-1980s. Four of the ten lead directly to RISC computers: the instruction set architecture (ISA) is optimized for the way HLL are used, the ISA is designed to be compiler generated, the ISA will not inhibit pipelining and instruction prefetching, and good compiler technology will be important in overall efficiency. The only one of the ten that really falls flat is the suggestion that there would be one ISA for languages like C and another for languages like LISP. The disappearance of microcode in machines of the late 1980s made such projections unlikely.

The paper ends with an impressive list of 52 references.

What happened to the authors

David Ditzel graduated from Berkeley and went to AT&T Bell Laboratories, where he was lead architect of the CRISP microprocessor, a RISC machine designed to run C programs efficiently. Its novel features included a Stack Cache, Branch Folding and unusually compact code size for a RISC processor.

In 1982 he started the first ASPLOS conference, which included the first paper on the IBM 801 RISC machine and the first public debate of RISC vs. CISC. He also helped start the Hot Chips conferences. He joined Sun Microsystems in 1987 and was Director of SPARC Labs where he helped advance SPARC processors.

In 1995 he founded Transmeta Corporation, where as President and CEO he is leading new innovations in computer design.

David Patterson got his tenure shortly after this paper appeared. and moved on to become department chair, and later SIGARCH chair and Computing Research Association chair. This paper described the first of a series of RISC-related research projects which produced several RISC chips: RISC-I, RISC-II, SOAR (Smalltalk On A RISC), and SPUR (Symbolic Processing Using RISCs), with RISC-I likely being the first VLSI RISC chip. In 1984 he started consulting with Sun Microsystems, which lead to SPARC being designed based on the various Berkeley RISC designs.

In 1987 Randy Katz and Patterson developed Redundant Arrays of Inexpensive Disks (RAID), which showed how to get more performance and higher reliability from secondary storage. To help make the more quantitative approach to computer design more popular, John Hennessy and Patterson co-authored two architecture textbooks in the early 1990s. Since that time he has worked on Networks of Workstations (NOW) with Tom Anderson and David Culler, which built large-scale systems from smaller systems using off-the-shelf switched networks, and most recently on Intelligent RAM (IRAM) with Tom Anderson and Kathy Yelick, which integrates a processor into a DRAM chip to provide a small, fast, energy-efficient computer for mobile, multimedia applications.

Retrospective:

Architecture of a Massively Parallel Processor

Ken Batcher

Dept. of Mathematics and Computer Science
Kent State University, Kent, OH 44242, USA
batcher@mcs.kent.edu

Having an opportunity to write a retrospective is a rare privilege: very few authors ever get a chance to comment on their earlier work. What follows are the various thoughts that came to mind when I read this 18-year-old paper again.

When the company asked me to write a paper on the MPP I decided to submit it to ISCA-7. Management balked at first: why pick a conference in La Baule, France when there were so many other conferences here in the U.S.? They only agreed after I convinced them that ISCA is the best conference in computer architecture; that the chance the paper would be accepted was very low; and that I could always submit it to some other conference if ISCA-7 rejected it. Management was much surprised when the paper was accepted: to help amortize my travel expenses they had me visit a few European groups that had some interest in the MPP so my trip lasted two weeks.

Readers can easily see what we thought of software at the time: the only mention of it is in a small paragraph in the section describing the PDMU. How naive we were to think that users just needed an assembler for PE micro-routines and another assembler for application programs! Fortunately NASA Goddard knew better and funded Anthony Reeves of Cornell University to write a Parallel Pascal compiler for the MPP: several scientists used the MPP at NASA Goddard and most all of them wrote their applications in Parallel Pascal.

At first the company wasn't going to respond to the request for proposals (RFP) from NASA Goddard: management thought it was useless to compete since other companies had much more experience building 2D-mesh-connected image processors. We on the technical staff thought we might win using what we learned about bit-serial word-parallel machines from STARAN. When management realized the staff was going to write a proposal anyway they gave up and let us submit it. It sure was a nice day for the staff when we won the contract!

The RFP specified only 256 bits per PE but we knew from STARAN that a bit-serial PE requires much more memory than that. STARAN also taught us what a pain it is to try and expand memory in an architecture that didn't allow for it. For the MPP we used 16 bits for every bit-slice address so ARU memory could grow to 65,536 bits per PE. SRAM technology at the time allowed us to supply 1024 bits per PE.

The RFP said the MPP should have "PDMU-ARU I/O Registers" so that's what we called the corner-turning memory. Soon after ISCA- 7 the name of the corner turner was changed to "Staging Memory."

The hardware design group in our department always followed very strict design rules and made sure that printed-circuit-board layers had the proper impedances and that all connections between boards were terminated properly. They were always proud to show me signals on the scopes: nice sharp pulses with no discernible ringing or other anomalies. Most likely the logic could have been run at a much higher speed but the S-RAM chips limited the clock to 10 MHz.

The project took about a year longer than expected because the PE chip went through two design cycles. The first design was silicon-on-sapphire but didn't meet specs. The second design met the specs using the advancements in bulk CMOS technology that were now available.

The machine was delivered to NASA Goddard on May 2, 1983: it met all specs and ran at the speeds shown in Table I of the paper. We never found any other customers for the MPP even though it was one of the fastest machines available at that time. Very few places had highly-parallel problems that could be spread out across 16,384 PE's so the market was very thin — lack of user-friendly software was another factor.

Many companies changed owners and names in the 80's. Goodyear Aerospace was sold to Loral and later Loral was merged with Lockheed-Martin so now the plant is known as Lockheed-Martin Tactical Defense Systems - Akron.

On October 29, 1996 NASA Goddard donated the MPP to the History of Technology branch of the Smithsonian. It's now resting in a warehouse in case some historian in the future wants to see what 1980 technology looked like.

The best place to find information on the MPP is "The Massively Parallel Processor", J. L. Potter, ed., The MIT Press, 1985. (ISBN 0-262-16100-1)

Today we would take a far different approach in designing a machine like the MPP. We would add some compiler and OS people to the team and try as much as possible to design the ideal parallel processor — a machine so user-friendly that most users wouldn't realize it's a parallel processor!

RETROSPECTIVE:

A Processor for a High-Performance Personal Computer

Ken Pier

Xerox PARC
3333 Coyote Hill Road, Palo Alto, CA 94304
pier@parc.xerox.com

Forward

I would like to thank the International Symposium on Computer Architecture for selecting "A Processor for a High-Performance Personal Computer" for their 25th anniversary anthology. The editors have asked for a retrospective on the history and genesis of the ideas — what influenced the thinking, why were certain decisions made, and the like, reported in this paper. By happy coincidence, I (Ken Pier) already wrote and published, in ISCA 10, "A Retrospective on the Dorado, A High-Performance Personal Computer" [1] with detailed answers to those questions. So I'm going to take the opportunity here to write a more historical, personal, anecdotal, and entertaining (I hope) addition to the description in [1], which may be found at http://www.parc.xerox.com/istl/members/pier/.

Prologue

In 1974, the year of ISCA 1, the revolutionary concept of personal distributed computing was well underway within the walls of the Xerox Corporation at its Palo Alto Research Center. What we now consider commonplace computing: personal hardware dedicated to the individual, with a high-resolution display, mouse input device, network connection to document viewing, filing and printing services, graphical user interface, and a wide variety of applications had all been created there in the space of a very few years. Outside of PARC, the focus was still on mainframes, time-sharing, minicomputers, and special-purpose computers connected together with special-purpose networks. IBM was King, the Seven Dwarfs [2] scrambled at its feet, and that seemed unshakable. Today, hundreds of millions of people make daily use of personal distributed computing, having never even heard that phrase.

However, as Alan Perlis and John R. White noted in their 1988 foreword to A History of Personal Workstations [3]: "*The transformation was not trivial, nor even inevitable. It was accomplished by people with energy, ambition, and purpose whose visions were supported by great technical expertise and insight. Their difficulties and triumphs are the stuff of which history is made.*"

Genesis

The Dorado was a proverbial "second system," successor to the Alto personal workstation. Three excellent papers [4, 5, 6] by Alan Kay, Adele Goldberg, Chuck Thacker, and Butler Lampson (first author of this anthology paper) describe the ideas and implementations that created personal distributed computing, the Alto, and its environment.

In brief, Alto was inspired by several visions. One vision came from Alan Kay in the early 1970s of personal dynamic media, to be embodied in a small, lightweight, interactive, personal device christened Dynabook. Dynabook would carry all the personal information, such as diary, calendar, documents to read and documents in the process of being written, painting and music compositions, animations, etc., for an individual. It would also contain a powerful computational engine to enable studying and experimenting with any system that could be quantified and simulated. It would be easy enough to use by a person as young as nine years of age. In short, it would turn computing completely around, making the machine the servant of the human. Although the Dynabook could not be built in 1970, Thacker and Lampson knew they could build a prototype not much bigger than four breadboxes and a television screen. And Kay and his team knew how they wanted humans to interact with machines — graphically, easily, and intuitively. Smalltalk, the first computing environment with a graphical user interface, would run on the Alto.

A second vision influenced the Alto, stemming from its corporate parent. Xerox PARC was commissioned in 1970 to develop the "architecture of information" for the electronic office of the future, to allow Xerox to play a major role in the emergence of that future, and to marry the Xerox expertise in copying and duplicating to that architecture.

While pursuing these and related visions, a new style of computing emerged, which became the mainstream form of computing today. As with many "paradigm shifts," it took nearly a generation from conception to full acceptance; when people encountered this radical, humanistic way of computing for the first time and "got it," it forever transformed their relationship to and way of thinking about human-computer interaction. Some did not get it. If you are young enough, some of the conventional wisdom of the 1970s may amuse you: that no ordinary office worker wanted or needed a computer at work, that executives would not learn to type, that machines that cost $15,000 in 1976 would never be cheap enough to be practical, and, of course, that no proud professional would ever let that "mouse thing" onto their office desk.

By 1975, just before I came to work for Xerox, hundreds of Altos, interconnected via Ethernet, were in daily use in Palo Alto, and a development department had formed out of the research center to capitalize on the electronic office of the future. It was clear that the first order of business was to create a much more powerful computer than the Alto, one capable of both offering real time response for users in the office setting and of handling the enormous data rates that high-speed office printing systems would need. A small team of engineers in the department, lead by Lampson and Thacker, set out to make such a system, basing its microtasking architecture on the Alto [5], but with greatly expanded IO bandwidth, virtual memory hardware, and support for modern programming languages which were also developed at Xerox [6].

Exodus

Events occurred rapidly even in the 1970s, and after eighteen months of preliminary work the department realized that the Dorado would be much too large and expensive to serve as a personal workstation. Meanwhile, the adjoining research center had bet its computational infrastructure on the Dorado for five or more years into the future. Thus, the Computer Science Lab decided there was no alternative but to assume responsibility for completing the task and the project was transferred from development into the research center. Ten or so scientists and engineers devoted some or all of their time over the next two years to specify, design, prototype, and launch manufacturing of Dorados for the PARC computer research laboratories, some giving up their research agendas for the good of the cause. The full story of this effort is detailed in [1]. I've selected a few hardware highlights to relate herein.

Numbers

The Dorado was to be blazing fast, compact, and capable of replacing Altos in the office environment, that is, to cohabit with humans. It was to evolve into the first "3M" machine: at least a million instructions per second from the processor, at least a million bytes of memory, and at least a million pixels of display. It was to occupy a box not larger than a four-drawer file cabinet. It was to be quiet enough to reside in an office, powered from an ordinary 110 VAC outlet.

Hardware parallelism provided the speed, and MECL-10K (a state-of-the-art, power-hungry MSI logic family, long before VLSI was invented) produced the blaze. The processor was capable of initiating a microinstruction every 60 nanoseconds (a "click") in production machines (prototypes were faster), completed most pipelined instructions in three clicks, and could execute simple macroinstructions in one microinstruction. A memory cache had a latency of two clicks, a throughput of a single click, and a hit rate on byte-coded instruction streams of nearly 99%. A hardware memory map, main memory storage, and an instruction fetch/decode unit all ran in parallel, independent of but coordinated by the microcoded processor. The processor was itself multiplexed to service all of the device controllers in the Dorado with no context switching overhead, due to hardware devoted to saving and restoring state for each individual device on demand. By having the computational power of the processor at their command, the controllers were simple, small, and fast.

To achieve compactness, a packaging system was developed in which high-density (288 MSI chips/board) custom-designed circuit boards slid horizontally into zero-insertion force connectors mounted in dual backpanels, which were really sidepanels. Boards were 0.625 inches apart. This meant that, unlike conventional machines, most of the circuitry would be inaccessible to oscilloscope probes for debugging; clever diagnostic software and brilliant deduction would have to suffice. In addition, the raw speed of the MECL logic and the need for controlled-impedance wiring made it impossible to use "extender" boards. Necessity may have been the mother of this invention, but it may also be due to the fact that Thacker could build or debug almost anything electronic, and Lampson, often explaining that "Science is Prediction," could debug complex circuitry in his head without benefit of physical aids.

But, a fully packaged Dorado with its large 300MB removable disk drive "... consumed 2500 watts of power, was the size of a refrigerator, and required 2000 cubic feet of cooling air per minute (while producing a noise level that has been compared to that of a 747 taking off)." [5, p. 285]. Only a slight exaggeration, but external packaging, cooling and noise reduction did not receive the level of attention required; we had no mechanical engineers on the Dorado team, and Dorados ended up rack mounted in machine rooms, cabled afar to offices and other workspaces where humans resided. Good thing, too.

Epilogue

By 1985 approximately 175 Dorados were in service. Although perhaps a dozen were marketed to laboratories and universities as the Xerox 1132, and a few tens exported to other sites within Xerox, the remainder were within a few hundred meters of one another at Xerox PARC. Dorados lasted until about the beginning of 1990 as the PARC workhorse workstation, when Sun Microsystems workstations, having been phased in over about two years, took over. The last Dorado was decommissioned in 1995 having served for over ten years as the PBX for an experimental digital telephone service, Etherphone [7].

In conclusion I would like to share a story and three favorite aphorisms from the history and evolution of personal distributed computing and the Dorado experience.

I presented this paper at ISCA 7 in 1980. It was my first technical paper and my first technical presentation. During the question/answer period following the paper, someone stood up and said, "How can you call a machine that costs $50,000 to manufacture a personal computer?" I was not prepared for such a question, but I remembered that I had chosen the title specifically to raise the bar on what should be expected of the personal workstation of the future. So I answered, almost without thinking, "The same way you can call a Mercedes a personal automobile." Perhaps "Ferrari" would have been more apt.

Lampson, quoting Browning in [6], reminds us that "a man's reach should exceed his grasp, else what's a heaven for?" Thus, customer requirements. L. Peter Deutsch informs us that "implementation is the sincerest form of flattery." Thus, commencement of system building. And the most important thing I have learned in my twenty-some years of building innovative hardware and software systems: nothing would ever be accomplished were it not for unwarranted optimism.

References

[1] Kenneth A. Pier, "A Retrospective on the Dorado, A High-Performance Personal Computer," *Proc. of the 10th International Symposium on Computer Architecture*, June 1983, pp. 252-269.

[2] The term was coined by the business and technical press of the time to denote the seven companies who attempted to compete with IBM. They were RCA, UNIVAC, Honeywell, Burroughs, Scientific Data Systems (SDS, later XDS, Xerox Data Systems), Control Data Corp. (CDC), and General Electric.

[3] A. Goldberg, Editor. *A History of Personal Workstations*, ACM Press, 1988.

[4] Alan Kay and Adele Goldberg, *Personal Dynamic Media.*, In [3], pp. 254-263.

[5] Charles P. Thacker, *Personal Distributed Computing: The Alto and Ethernet Hardware*, In [3], pp. 267-289.

[6] Butler Lampson, *Personal Distributed Computing: The Alto and Ethernet Software*, In [3], pp. 293-335.

[7] Douglas B. Terry and Daniel C. Swinehart, "Managing Stored Voice in the Etherphone System." *ACM Transactions on Computer Systems 6(1)*, pp. 3-27, 1988.

Retrospective:

Lockup-Free Instruction Fetch/Prefetch Cache Organization

David Kroft

14 Kings Inn Trail
Thornhill, Ontario
L3T 1T7

How does one begin to describe the dreams, thoughts and fears that surround a discovery of a different view of some old concepts or the employment of old accepted methodology to new avenues? It is probably best to start the account by describing the field of Computer Architecture, in particular, the area of hierarchical memory design, that was prevalent around and before the time the ideas came to light.

In the mid seventies, I was fortunate to be selected as one of the members of a design team to design and develop the central processing unit (CPU) for a low end model of Control Data's new line of main frame computers. Since integrated logic circuit components were now readily available and, consequently, computer hardware was much cheaper, the new line would feature a highly expanded instruction set — the move in vogue at that time was toward complex instruction set computers (CISCs). Note, that due to the price of hardware, all former CPUs were of the reduced or minimum instruction sets varieties (RISCs). The fall of the cost of hard logic and memory, also, allowed for the implementing of the new hierarchical memory design concepts into these next computers to be designed, developed and manufactured. Control Data, or should I say, the technical visionaries of Control Data at that time had differing opinions as to the advantages of hierarchical memory design. I recall one of these visionaries telling me the following: "We, at Control Data, have the know-how to design central memories big enough, fast enough and put them close enough to the CPU that memory hierarchy would never be needed." I was, however, able to convince at least a sufficient number of these technical gurus that a Cache Memory would be advantageous to the new CPU so that I was given the assignment to design the Cache for this low end model of the line. The fact that IBM was now incorporating Caches in all their new designs, obviously, helped me considerably with my persuasion. Note that there were other later computers designs at Control Data that did not have Caches — believe it or not.

So, there I was in the mid seventies, having never designed anything real up to then — I had accomplished a lot on paper — given with the assignment to solely perform the system design and logic design implementation of a Cache for this new low end model with its extended instruction set. I say, solely, because no Cache had ever been included in any of the Control Data designs previously and there were a number of "Doubting Thomases." Due to the opposition or possibly since I had still had the courage and adventure of youth — I was only in my early thirties, I embarked on a very ambitious design approach.

First, I included most of the latest Cache concepts that were contained in the literature. I decided on a set associative organization with 128 sets (rows) of four or eight blocks (lines) per set and a block size of 32 bytes (4 8-byte CDC words). In addition, write through and a least recently used (LRU) replacement algorithm were chosen. These selections were made after much study and consideration of the simulation data in the literature and with the restrictions imposed by the then available memory components (a fast 256 by anything was not on the horizon while a fast 128 by 1 was there.) The two set sizes (columns), (4 and 8), allowed for the two required Cache sizes of 16K and 32K bytes. In retrospect, the complexity (i.e. additional cost) of the additional associativity for the marginally higher hit rate for the larger Cache was the only decision I regret.

Next, I decided to incorporate the additional cost of having the Cache's address space be an unique virtual address space thereby putting the cache closer to the processor; the translate lookaside buffer (TLB) would now operate in parallel with the Cache rather than before it. Note that the Cache only needs the real memory address from the TLB on a miss. Lastly, just before the miss overlap feature (from paper) inclusion in the design, I discovered a way to allow for the graceful degradation of the Cache via the disabling of any one of the four 8K bytes sections at a time. The above required just the need to determine the maximum LRU of the enabled blocks in a set when block replacement was needed.

Recall that this new line of machines envisioned by Control Data had an extended instruction set. In fact, the instruction set included almost all the instructions of all formats that were present in one machine or another. Many of the instructions made more than one reference to memory. At first, I thought, how can one order the memory references so that the first references always produce a hit in the Cache to prevent impact on the following ones. At the same time, I saw that the Cache had effectively two input ports, one from the execution unit and one from the instruction unit. Why should the hit/miss of one port impact the other, I reflected? I was told that a two port Cache with each port totally independent of one another was an open problem with no solution known as yet and the route most manufacturers were taking to circumvent this two port problem was to have two Caches, one for each unit — these units, it was known, each had different classes of data. There were, now, two cases for miss overlap or for Cache hit(s) being processed while a Cache miss was outstanding. How can this be accomplished?

I associated the above problem with everyday occurrences. I pondered "what if" there was a queue of people waiting for service from one particular individual and this individual could service these people either with one of two possible scenarios - a long time consuming service due to the need for additional parts, information, etc. or a short quick response. The solution for minimum wait time would not be solved by reordering the people in the line; a second individual is required. How should this second individual help? The second individual would back up the first by going to get the necessary parts, information, etc. while the first continued to service the queue. The above thought process permitted me to see a vision for overlapping misses. I will save all the relevant information about a miss, forward a request to a "block getting unit" for the block needed and then, continue processing the input service requests. A block receiving unit will interrogate the relevant information about a miss, forward the response to the waiting individual and update its own unit, the Cache buffer, if advisable. The block getting unit and the block receiving unit are this second individual. A light has come on; the long process of determining all the details and workings must now be done to determine viability. Fortunately, all fell into place, if not initially, then during debug.

As indicated at the beginning, a discovery is just a different look at some old concepts or the use of some old concepts for new mechanisms. Mine was the latter.

Retrospective:

A Study of Branch Prediction Strategies

James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
jes@ece.wisc.edu

In 1979, I took a leave of absence from the University of Wisconsin to work at the Control Data Corporation in the Twin Cities. In truth, my intention at the time was to abandon an academic career; I felt awkward teaching computer design — never having worked on the design of a computer. And I was doing research in fault-tolerant computing as a theoretical enterprise, which seemed contradictory.

The project I joined at CDC was developing the high end processor of a new product line — the Cyber 180 series. The processor was code-named "Theta", and would become the Cyber 180/990 when it was officially announced. The Cyber 180 architecture was a 64-bit virtual memory system with all the bells and whistles that were fashionable at the time. It was built around register-register instructions, but also had a number of complex instructions to support a heavy duty protection system, business data processing, and a memory-to-memory vector instruction set. With the same hardware, it also had to support the older 60-bit Cyber 170 instruction set, and switch seamlessly between the two modes — potentially at the procedure call level.

My manager was Jim Stockard, and I took technical direction from Ron Hintz, the chief architect. At the time I joined, the project had been underway for awhile, and was beginning to fall short of performance goals. My main responsibility was to carry out performance studies and suggest performance features to the designers.

The technology was based on ECL gate arrays — about 200 gates per chip. It had been developed for the Cyber 200 supercomputers, which ran at a 20 ns clock cycle. The Theta clock cycle was 16 ns, however, and in retrospect, this was probably a touch too aggressive. The complex instructions, aggressive clock cycle, and the packaging technology led to very long pipelines. It took about six pipe stages to fetch an instruction, decode it, and generate what were called "micrands", ready to be issued (this process was very similar to what is done today in aggressive x86 implementations). Instructions issued in order, at most one per cycle.

With a pipeline this long, conditional branches were a performance problem. The original pipeline design used a very simple static prediction strategy where all branches were predicted taken (or maybe not taken — I don't recall which). Instructions following a branch had to wait for the branch to be resolved before they could issue.

During discussions with Tom Lane, the designer responsible for part of the instruction pipeline, the possibility of branch prediction came up. Tom had done a small study, using a cache-like table with single-bit entries. Tom also pointed me to Shustek's thesis [1] — a real classic that suggests a number of static prediction strategies.

I considered several prediction methods, most of which are given in the paper. It was evident that dynamic prediction was better than static. For studying performance, most of the benchmarks I was using (kernels, actually) were heavily loop-dominated. It was evident that using a single history bit led to two mispredictions at loop terminations. Solving this problem with a two-bit saturating counter seemed like a good thing to do, and simulations showed a performance advantage. The parts available for implementing the predictor table were 16 x 4 register file chips, so a two-bit table entry cost no more than a one-bit table. I studied larger counters, but two bits worked better than three or four.

Hardware was at a premium, and early on I realized that the table could be indexed like a cache, but tags were unnecessary — there was already a way of recovering when mispredictions were made. I believe the final implementation consumed four 16x4 register chips (a 64 entry table) and a single gate array for updating. The counters described in the paper used two's complement arithmetic with the sign bit being used to indicate taken/not taken. However, the final implementation used the now-common 0-to-3 integer counter.

When I joined the Theta project, I took over the performance simulator from Paul Higgins; it had been developed by Paul and Dick Olson of CDC in Canada. The simulator was written in ASPOL and was trace-driven. We had a pile of 9-track trace tapes — including both Cyber 170 and 180 codes. The benchmarks were FORTRAN kernels and one nasty system benchmark called the SWL-profile ("swill" for short).

The benchmarking in the paper was very basic by today's standards — yet the paper made it into the "Performance Analysis" session at the conference. Many of the kernels were only a few lines of code. I included the more difficult-to-predict kernels, but even those were pretty small. In doing the study, the SWL-profile was also simulated, and it showed the biggest performance improvement, but these results were not included in the paper.

In conjunction with branch prediction, the actual Theta design also included a scheme for "conditional issue" ("speculative execution" today). As mentioned above, instructions issued in order, and branches took 5 cycles to resolve (only some simple integer instructions were faster). When a branch issued it reserved all result bus slots up until the time it finished. Then, following instructions could conditionally issue and start executing prior to the resolution of the branch. The reserved bus slots inhibited any conditionally issued instruction from writing its result register before the branch was resolved. A branch misprediction would invalidate all conditionally issued instructions and start over.

Overall, with branch prediction and conditional issue, the FORTRAN benchmarks improved by about 5%, and SWL about 10%. The SWL improvement was the real clincher, because the Theta project was furthest behind in SWL performance goals.

This work was done in the context of a development project, so there wasn't much time for academic investigation. The only exception is the small study I did using larger counters and assigning confidence levels depending on the count values. The concept was to speculate more aggressively when the confidence level was higher. I continued to carry this idea around for about 15 years before working on the study that appeared in Micro-29 [2], two years ago.

This was my first ISCA paper; the conference was held in the Twin Cities, which provided additional motivation for writing the paper. One of my clearer memories of the conference is the invited speech Jim Thornton gave. I also recall meeting David Kroft, another CDC employee who developed a non-blocking cache for a different Cyber 180 machine being designed in Canada.

A number of Cyber180/990 machines were eventually built and sold; I believe several went to Europe, where CDC did a good business at the time.

References

[1] L. Shustek, *Analysis and Performance of Computer Instruction Sets*, Ph.D. Thesis, Stanford University, 1978.

[2] E. Jacobsen, E. Rotenberg, J. E. Smith, "Assigning Confidence to Conditional Branch Predictions", *29th Int. Symp. on Microarchitecture*, Dec. 1996.

Retrospective:

RISC I: A Reduced Instruction Set Computer

David A. Patterson and Carlo H. Séquin

Computer Science Division
University of California, Berkeley, CA 94720
{pattrsn,sequin}@CS.Berkeley.EDU

This 1981 paper was written as part of the RISC movement that began to flourish in the early 1980s. The three groups leading the charge were at IBM, Berkeley, and Stanford.

IBM was the earliest, focusing on advances in compiler technology and instruction sets that compilers could use to get good performance without the need for a microcode interpreter. Their targets were a 24-bit ECL minicomputer for hardware, called the 801, and a programming language they invented called PL8, and their competition was the IBM 370 family of computers.

As the introduction to this paper suggests, the Berkeley effort was in trying to design an instruction set that made sense for a single VLSI chip. Our group did not include compiler experts, so that was not something that we were pushing. Our targets were a 40,000 transistors, 32-bit NMOS microprocessor and the programming language C and UNIX operating system, and the competition was the VAX-11/780, a relatively new machine that was making big waves in the marketplace.

The Stanford effort was also interested in a 32-bit single chip microprocessor, called MIPS for Microprocessor without Interlocked Pipeline Stages, and since Hennessy knew compilers they pushed it as well. They concentrated on the Pascal language, and while they didn't typically compare to other machines, occasionally they compared to the PDP-10.

The Berkeley RISC effort was inspired in large part by Patterson's reaction to a sabbatical he took at DEC in Fall 1979, and by our goal to make our architecture courses "hands-on" and as relevant as possible. This was the first time a university planned to actually build a complete microprocessor on a chip, and many people let us know that we had almost zero chance of success. So we were well aware that we had to keep the structure and the logic of this chip as simple as we could get away with. Séquin, at that time, was involved as a consultant in the Mead-Conway revolution of getting universities involved in chip design. Having previously built several chips at Bell Labs, he was more aware of what it would take to make a working chip, but tried to hide his anxieties in order not to dampen the enthusiasm for the project.

Patterson had worked on microprogramming tools for his Ph.D., and that was what he had been helping with at DEC. He wondered about building a VAX as a single chip, especially given all the microcode bugs which were often patched in the field. (Indeed, IBM invented the floppy disk just to have a convenient media to ship patches to microcode.) Upon his return to Berkeley, he submitted a paper to IEEE Computer saying that the only way to build a VAX-class machine on a chip, with all its complexity in microcode, was to provide mechanisms that would allow patches to occur. The paper was rejected, as reviewers said such a technique was wasteful of silicon resources, which was certainly true, but it was also in Patterson's opinion not possible at the time to get the VAX microcode perfect in order to mass produce chips. If both positions were valid, then perhaps the solution was to re-examine the value of the instruction set complexity in the VLSI age?

The Berkeley work was done as part of a series of graduate classes, and the early statistics in the paper were generated by the first class investigating the RISC ideas, starting in January 1980. This series of courses included learning Mead-Conway design, investigating the RISC architecture ideas, and then implementing RISC-I. Many Berkeley students took some of the courses, but students who stayed all the way to the end included Dan Fitzpatrick, John Foderaro, Manolis Katevenis, Jim Peek, Bob Sherburne, and Korbin Van Dyke.

John Cocke of IBM visited for a day during those courses and gave us a very inspiring endorsement of our plan to keep the instruction set as simple as possible. Sometime that winter we also heard a talk by Forest Baskett in which he expressed his desire to have a really large number — e.g., "one thousand" — registers. Some time later, after we had completed the course series, Peter Denning came to give a "qualitative" architecture talk in Spring 1980, and he found an audience loaded with statistics as well as opinions about the value of complex architectures. The following year or two Nick Tredennick presented us with a seminar with the provocative title "Why RISC is a pile of junk." So these were rather exciting times!

One story of interest is where the name RISC came from. In the winter of 1980 Dave Patterson and Carlo Séquin were driving to Silicon Valley to go to a program committee meeting, and they were talking about what to name the project and where to get research funding for the project. Our hope was to get funding from the Department of Defense Advanced Research Project Agency, as DARPA was the prime agency for providing sufficient budgets to actually build chips and systems. The agency's motto was to fund "high risk, high payoff" proposals, as DARPA's responsibility was to push the state of the art to make sure the United States was not surprised technologically as it was by the Sputnik satellite in 1957. Hence we decided to name the project RISC since by definition it was "high risk", hoping that DARPA would see the high payoff and support our work!

One thing about the RISC vs. CISC debate is the revisionist history on what was CISC. The paper says the trend towards complexity was given by comparing VAX vs. PDP-11, iAPX-432 vs. 8086, System 38 vs. System 3. The VAX and the 432 were the ones that we questioned, and we think those concerns hold up pretty well today. The 8086 may have been inelegant, but it was not particularly complex, and this paper used it as the example of a simple machine. I think the trade press concluded that any commercial computer that wasn't a RISC must be a CISC, and hence the confusion.

Something to keep in mind while reading the paper was how lousy compilers were of that generation. C programmers had to write the word "register" next to variables to try to get compilers to use registers. As a former Berkeley Ph.D. who started a small compiler company said later, "people would accept any piece of junk you gave them, as long as the code worked." Part of the reason was simply the speed of processors and the size of memory, as programmers had limited patience on how long they were willing to wait for compilers.

This brings us to one final story about RISC a few years later. The register windows of RISC-I were there to make sure the operands stayed in registers versus in memory, as a RISC machine that keeps scalar variables in memory is a slow computer. When Patterson was consulting for Sun on SPARC, the Sun compiler expert was Steve Muchnick. Muchnick asked Patterson about register windows for Sunrise, the code name for SPARC. Patterson said variables need to be in registers, so the question was whether the Sun compilers were going to implement the graph coloring algorithm from IBM that did a very good job of register allocation. Muchnick got a ghastly look on his face, and gave an emphatic NO. Patterson said then sunrise better use register windows. Two years later when the machine shipped Sun's compilers had improved such that graph coloring was not such a ghastly prospect, but by then the die had been cast.

Looking at the technical content of the paper, the definition of RISC-I stands as a pretty good definition of RISC machines and the RISC-I instruction set is still a very reasonable 32-bit integer instruction set — given that you want to build the whole processor in less than 50,000 transistors. Now that we know about the 801, you can see the differences in the instruction format (16-bit immediate field vs. 13-bit for RISC-I) and terminology ("execute and branch" vs. "delayed branch".) The paper claimed the RISC-I code size was about 1.5 times the VAX, and that is probably still a reasonable estimate. Claiming that a single chip computer was faster than the best selling minicomputer was a bold claim, even if the claim was qualified by saying it was made on only using two small programs. Bhandarkar and Clark [1] did a careful study 10 years later when there were real systems to compare, and reached the same conclusion: "RISC as exemplified by MIPS offers a significant processor performance advantage over a VAX of a comparable hardware organization."

What have the authors and key students been doing since this 1981 paper?

David Patterson got his tenure shortly after this paper appeared and moved on to become department chair, and later SIGARCH chair and Computing Research Association chair. This paper described the first of a series of RISC-related research projects which produced several RISC chips: RISC-I, RISC-II, SOAR (Smalltalk On A RISC), and SPUR (Symbolic Processing Using RISCs), with RISC-I likely being the first VLSI RISC chip. In 1984 he started consulting with Sun Microsystems, which lead to SPARC being designed based on the various Berkeley RISC designs. In 1987 Randy Katz and Patterson developed Redundant Arrays of Inexpensive Disks (RAID), which showed how to get more performance and higher reliability from secondary storage. To help make the more quantitative approach to computer design more popular, John Hennessy and Patterson co-authored two architecture textbooks in the early 1990s. Since that time he has worked on Networks of Workstations (NOW) with Tom Anderson and David Culler, which built large-scale systems from smaller systems using off-the-shelf switched networks, and most recently on Intelligent RAM (IRAM) with Tom Anderson and Kathy Yelick, which integrates a processor into a DRAM chip to provide a small, fast, energy-efficient computer for mobile, multimedia applications.

Carlo Séquin was made Computer Science Division chair in the fall of 1980, and was thereby yanked out of the mainstream of the development of the successor chips to RISC I and II. When he returned to research full-time, after an influential sabbatical at Siemens Corp. in Munich, Germany, he decided to abandon chip building in favor of building tools that would ease the tedium of designing and debugging VLSI chips. Together with Richard Newton and Alberto Sangiovanni-Vincentelli they launched the Berkeley Synthesis Project which focussed a large number of faculty and graduate students on an effort to build CAD tools and integrated circuits in a symbiotic manner. Towards the end of the 1980's Séquin wanted to build CAD tools that would involve more than just the two dimensions used in chip layout. He had had a long-standing interest in Computer Graphics. The design and construction of Soda Hall, the new home of the CS Division since 1994, provided him with the opportunity to create a complete computer model of Soda Hall and to develop tools for the interactive exploration of virtual buildings. Since then he has worked on the development of CAD tools for architects and for mechanical engineers, and most recently even started to collaborate with sculptors who would implement his computer-generated artistic designs in wood or in bronze.

Dan Fitzpatrick got his Ph.D. in 1983, working on Computer Aided Design software for VLSI. He initially joined VLSI Research.

John Foderaro also got his Ph.D. in 1983, but his interest was in symbolic manipulation systems. He helped found a Berkeley software startup called Franz, Inc, which initially created a LISP programming environment. He still works for that company.

Manolis Katevenis was the lead graduate student on the project, and his 1983 dissertation won the ACM Distinguished Dissertation Award. He went on to work as a faculty member at Stanford university for a year before returning home to Greece. He is now with the University of Crete, Dept. of Computer Science, where he is currently Professor and Associate Chairman. Since 1985, he is also with the Institute of Computer Science, FORTH, Heraklion, Crete, where he is currently the Head of the Computer Architecture and VLSI Systems Division. His recent work has been on very fast switches (see http://www.ics.forth.gr/proj/avg.)

Jim Peek came to Berkeley for an MS, and has since worked for a variety of computer companies in Silicon Valley, and most recently was working at Sun Microsystems.

Robert Sherburne got his Ph.D. in 1984 and went to work originally at what became Pixar to build graphics hardware. He has since joined Silicon Graphics. Katevenis and Sherburne built RISC-II, which was probably the first microprocessor from a university accepted for publication at the prestigious International Solid State Circuits Conference.

Korbin Van Dyke also came for an MS, and joined an early 8086 clone company that was eventually purchased by AMD. He now works for Chromatics Research.

References

[1] D. Bhandarkar and D. W. Clark, "Performance from architecture: comparing a RISC and a CISC with similar hardware organization." *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.310-19, April 1991.

Retrospective:

Decoupled Access/Execute Architectures

James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
jes@ece.wisc.edu

In early 1981, I was still at Control Data in the Twin Cities. The Cyber 180/990 project was close to the prototype stage, so no new performance features could be added. It was apparent there wasn't much left for me to do. Also, there were a number of interesting problems that had come up during my year and a half at CDC, and time hadn't allowed pursuing them as much as I had wanted — returning to the University of Wisconsin would provide the opportunity. I decided to resume my academic career, this time in computer architecture, and in late May I headed back to Madison.

The Cyber 180/990 had issued instructions in order, at most one per cycle. And, within CDC at the time, these were treated as fundamental constraints. I had some rather vague notions of how to overcome these barriers — but left CDC more with goals in mind than any specific solutions.

Because my prior experience had been with numerical problems, back at Wisconsin I hit upon a way of achieving multiple issue and dynamic scheduling with two instruction streams and queues. One instruction stream was for addressing and one for computation. Each stream would issue in order — maintaining simplicity. I remember being pretty excited about the novelty of the concept — but was a little deflated a few weeks later when I read about the CSPI array processors in the Sept. '81 issue of Computer Magazine [1]. These weren't general purpose computers, but used basically the same access/execution decoupling. After the ISCA paper appeared, I also became aware of the SMA work that Andy Pleszkun had done for his Ph.D. with Ed Davidson at Illinois [2]. And, about a year later at a workshop in New Orleans appeared yet another machine with similar concepts [3]. It was a proposed machine called FOM (FORTRAN Oriented Machine) from IBM — a place where superscalar concepts had been kicked around for a long time.

The benchmarking in the paper was pretty miserable by today's standards. I used compilations for the Cray-1 as a guide. The actual simulations were done by hand, and average speedups were calculated as, *ahem*, the arithmetic mean.

The simplicity of in-order instruction issue had been drilled into me at CDC — in retrospect, too much. It probably prevented me from looking at more flexible superscalar machines early on. It is my observation that a common mistake of architects has been (and continues to be) overestimating the complexity of dispatch/issue logic.

I still think the idea of two separate instruction streams connected with branch queues was neat. And having two PCs helped with the precise interrupt problem. But later in a study published at a small conference, Tom Kaminski and I looked at a scheme that combined instruction streams in the binary and had a hardware "splitter" that divided the stream after it was fetched [4]. This was the form that showed up later in the ZS-1 [5] (another, longer story). With this modification, decoupled machines were similar in appearance to the first IBM RS/6000s. A major difference is that the decoupled machines use architectural queues for renaming memory operands. This had the advantage of renaming the values that were most important — load values, and the queue discipline made management of the physical locations very straightforward.

Following this paper, the research got a big boost when Shlomo Weiss came along. Building on tools Nick Pang had developed, Shlomo did substantial performance studies (and he and I realized in the process that harmonic mean speedups should be used). These more detailed results appeared in the IEEE Transactions on Computers a few of years later [6]. Along the way, Honesty Young also added significantly to the Cray-1 simulation tool set which benefited this research. After I left the University in 1983 to work on the ZS-1, research on decoupled architectures at Wisconsin continued with the PIPE project [7], headed by Jim Goodman, Andy Pleszkun, and Randy Katz. The PIPE project produced a number of significant papers on decoupled architectures — including one that appeared in the 12th ISCA [8].

References

[1] E. U. Cohler and J. E. Storer, "Functionally Parallel Architecture for Array Processors," *Computer*, vol. 14, no. 9, pp. 28-36, Sept. 1981.

[2] A. R. Pleszkun, *A Structured Memory Access Architecture*, Computer Systems Group Report CSG-10, Coordinated Science Lab., Univ. of Illinois, Urbana, IL, Oct. 1982.

[3] W. C. Brantley and J. Weiss, "Organization and Architecture Trade-offs in FOM," *IEEE Workshop on Computer Systems Organization*, New Orleans, pp. 139-143, March 1983.

[4] J. E. Smith and T. J. Kaminski, "Varieties of Decoupled Access/Execute Computer Architectures," *20th Allerton Conference on Communication, Control, and Computing*, Monticello, IL, pp. 577-586, Oct. 1982.

[5] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon, "The ZS-1 Central Processor," *Second Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, pp. 199-204, Oct. 1987.

[6] J. E. Smith, S. Weiss, N. Pang, "A Simulation Study of Decoupled Architecture Computers," *IEEE Transactions on Computers*, Vol. C-35, pp. 692-702, Aug. 1986.

[7] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman, "PIPE: A High Performance VLSI Architecture", *IEEE Workshop on Computer Systems Organization*, New Orleans, pp. 131-138, March 1983.

[8] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI Decoupled Architecture," *12th Int. Symp. on Computer Architecture*, pp. 20-27, June 1985.

A Personal Retrospective on the NYU Ultracomputer

Allan Gottlieb

New York University and NEC Research Institute
New York, NY 10003 Princeton, NJ 08540
gottlieb@nyu.edu gottlieb@research.nj.nec.com
http://allan.ultra.nyu.edu/gottlieb

Introduction

The NYU Ultracomputer project, a long lasting research endeavor, was started in 1979 by Jack Schwartz and was very active throughout the 80s and the early 90s. This project was an early attempt to explore the possibilities of large scale, shared-memory parallel computers. The project was broad spectrumed: we developed hardware primitives for coordination and implemented full-custom VLSI chips to speed their concurrent execution, we built prototype multiprocessors incorporating these chips, we implemented a scalable, symmetric Unix operating system, we ported compilers to several microprocessors, we contributed to network design and analysis, we implemented a parallel lisp system and worked on prologue, and we developed algorithms and software for scientific applications.

The 1982 ISCA paper you have before you, a revision of which appeared in the Feb. 1983 "*IEEE Transactions on Computing*", includes contributions in three areas: first, the coordination primitive fetch-and-add and its generalization to fetch-and-phi; second, a technique inspired by Larry Rudolph for combining in hardware concurrent memory references, including concurrent fetch-and-adds, directed at the same memory location, and a high-level VLSI design, inspired by Marc Snir and Jon Solworth, for network switches implementing combining; and third, analytic results primarily due to Clyde Kruskal and Marc Snir on the performance of buffered multistage networks. Fetch-and-add is now present on commercial processors including models from SGI-Cray and Intel. Hardware combining inspired a mini-industry of research results from a number of institutions, but has not been realized commercially in anything like the generality we proposed it. The analytic network results form part of a well developed theory with many contributors.

In the mid 80s, we cooperated closely with a team from IBM research in the development of their RP3 system. This cooperation raised our fame (and funding level) significantly and had several other very positive results. There were also some imperfections in the cooperation as discussed below: one in particular highlighted a weakness in the NYU software team in general and Allan Gottlieb in particular.

Our first compilers were PCC-based and targeted the 68K microprocessor used in our early hardware. When we decided to use the AMD 29K for the Ultra III prototype, we chose GCC as the compiler and we, primarily Richard Kenner, retargeted it to the 29K and to the IBM ROMP used in the IBM RT/PC workstations that constituted our development environment (and that were used in the IBM RP3). Kenner became very interested in GCC, retargeted it to other microprocessors, and has been the lead developer of its machine independent portion for the last several years, an important "spin-off" of our research efforts.

I summarize a few accomplishments in the next section. Since we have bragged before and many of these boasts can be found starting at my home page, I have kept the next section short. Less easily found in the literature is an account of our shortcomings, which is the subject of the longer section thereafter. Finally, I offer some thoughts on doing it again.

Accomplishments

The primary achievement is that, at the end, it all worked. We, primarily Ron Bianchini, built Ultra III, a 16-processor Ultracomputer prototype for which we designed the circuit boards. Ultra III featured custom VLSI components, designed primarily by Susan Dickey and Richard Kenner, that combined concurrent memory references, including various fetch-and-phis, directed at the same location. Our Symunix operating system, due primarily to Jan Edler, ran well and all the code was compiled using our port of GCC. In the 80s we had built several bus based systems that ran an earlier version of Symunix. These Ultra II prototypes were used successfully by researchers both within and outside NYU and also by NYU students taking courses in parallel computing. Their reliability was outstanding for the time: I would say each Ultra II crashed (including hardware and software problems) about once a month during its heyday.

Running a complete system with hardware combining chips gave credence to our claim that support for combining does not significantly increase network latency during periods when no combining occurs. Indeed for the systolic queues we implemented, combining was not on the critical path. While this last statement is implementation and technology dependent, I believe that any similar implementation technique would also show little or no speed penalty for combining. I also believe, but this is much softer, that, independent of combining, systolic queues are a competitive design for any buffered network. The area and pinout penalties for combining are factors of 2 and 1.5 respectively. Since network chips in general, and ours in particular, are pin limited, I estimate that combining adds 50% to the network cost. However, if both directions of a combining switch fit in a single package, there are no extra pins needed and the hardware cost of combining is small.

I would include in accomplishments the friendship and good will that characterized our group both during times of lavish funding and during the contraction period that followed. I cannot recall a single instance where there was a significant argument on a *personal* level. (Like many groups, we had constant technical and scientific disagreements, often quite animated — the NY does stand for New York, but the antagonists remained friends throughout.)

Regrets

There are many changes in history that I could propose, which would have increased our accomplishments. For example, if Jan Edler and I had not fathered children during the period, we would have developed more software — but these are not regrets. Instead, I will only discuss those where we should have done better (even if I am not sure what exactly we should have done).

No Symunix II

I am very proud of Symunix. I am also very proud of Jan Edler's thesis, which should have become Symunix II. Instead, only parts were implemented and it never went into full production. Perhaps due to our heads being too swelled by the success of Sym I, we were too ambitious with Sym II and never stopped improving the design. One of the team members complained that she couldn't implement as fast as the design changed, so the end of tunnel kept receding. I believe such over ambitious behavior is referred to as the "second system syndrome".

No Hardware, No Software

Sym II was to be hosted on both our Ultracomputer and IBM's RP3. After work had begun on Sym II, IBM placed an RP3 simulator and a 4381 host in our laboratory for us to use. However, we never made full use of the simulator. Instead most Symunix progress continued to occur on our prototype hardware. Although this observation sounds like a clear NIH problem, it is not that simple. IBM also gave us "pseudo" RP3 hardware that had the same processor as the RP3 but a different memory management unit and was not software equivalent in other ways. Our operating system task on this machine was accomplished successfully showing that we could write OS code for NIH machines. The shortcoming we exhibited was that we just couldn't get excited about writing for a simulator. If we didn't have hardware (even flaky hardware, like our prototypes during hardware bring-up, was good enough), we somehow couldn't produce software at anything close to our normal rate. This was a clear failing of our very talented software group, with primary blame falling on its leader, me.

No NYU Switches with IBM Technology

The section title may be somewhat misleading. IBM worked on a different combining technique for the RP3 but canceled this VLSI effort, which had no direct NYU involvement, prior to the availability of working chips. The regret that I am referring to, however, is our own effort to port the "*already working*" NYU VLSI design to IBM technology. We spent considerable effort over a prolonged period but could never overcome the hurdles we encountered. The design rules were proprietary, were different in kind from what we had dealt with, and we were not permitted direct access to the design rule checker. Instead we had to convert our VLSI description files to another format, which we then emailed to IBM. When the design rule check was complete, the results were emailed back. Now a dozen years later it sounds like just a few months work but we never succeeded despite trying hard.

The Blankety-Blank Miswired Wirewrap Boards

The Ultra II boards were wirewrapped and worked quite well. For Ultra III we used a larger form factor and some pin grid arrays but the wirewrap vendor said it was no problem. We had one board of each type (PE, MM, SW) wrapped and indeed they worked after a few mods. We then had the full complement of 4 PEs, 4 MMs, and 8 SWs wired and *none* of the 16 was close to working. Our NSF site visit was two or three months off.

One month later, we still had no working boards, went into frantic mode, and started to find the problems. *Many* of the wires were pulled too tight and broke. Naturally the insulation did not break so the wires looked fine. Since we had paid for continuity checking, considerable impolite language was uttered. But this was the easy part. *Many* other wires had been bent hard around pins and thus developed intermittent shorts. What a mess! We managed to get a partial system sort of running for the NSF team and the visitors, who had been through these wars themselves, understood and helped us get the vendor to rewire the boards at its own expense. Although still poor, the new boards were better. But it took us well over a year to fix them.

In retrospect, the success of Ultra II made us too ‘‘comfortable’’ with wirewrap. We should have realized that we were pushing that technology too hard and that vendors were becoming less familiar with it. In short Ultra III should have used printed circuit boards as was done for 2nd generation Ultra II MMs. I must point out that all the junior members of the hardware group were pushing for PC boards; the old folks like me overruled them.

Play it Again Sam?

Professor Gottlieb, looking back would you do the following things again?

- Build Hardware?

There are certainly downsides to building hardware, especially when you don’t have an engineering school: The effort is very labor intensive and depends on the kindness (or at least competence) of strangers (in our case the wirewrap vendor). On a positive note we did get to experience the thrill of victory... but sadly only after repeated agonies of defeat. Despite the difficulties of building hardware, it really was the only way to answer the key questions about combining and fetch-and-add. If I knew then what I know now, I would do it again, of course somewhat differently. However, building a full system today is a more problematic decision. The bar has been raised rather high and I realize that designing boards with modern signal speeds might well require more engineering prowess than a non-engineer PI at a non-engineering university is able to provide.

- Work with IBM?

As mentioned above, we spent considerable effort in an IBM-related VLSI effort that bore no fruit and we performed poorly with their simulator. In addition, it was ruled impossible to have the RP3 OS based on Version 7 of Unix. As a result Sym II moved to BSD4.2, which added to the former’s complexity in ways not related to parallel computing. Nonetheless I am pleased we worked with IBM. For me personally it lead to a long lasting friendship with George Almasi, one output of which was our book ‘‘*Highly Parallel Computing*.’’ The cooperation also forced us to see how our ideas could be used in a system that was not our own and how to compromise with outsiders, both positive consequences. We also benefited from the close contact with the RP3 developers. In particular, they suggested improvements to the Sym II design. Most importantly, for many people outside of Greenwich Village in NYC, the RP3 collaboration put us on the map with clear recognition and funding advantages.

- Shun Mach?

It was suggested to us by ARPA that we incorporate our parallel algorithms in the Mach OS rather than continue with Symunix. IBM was agnostic on the OS issue technically but preferred that we consent to ARPA’s request. When Mach became available on the RT/PC workstations in their offices, the IBM preference became stronger. In addition to potential funding advantages, our switching to Mach would have made us a player in what was then deemed likely to become a major OS. Nonetheless we refused. We were not agnostic on the issue technically; instead, we believed strongly that our approach was better for large-scale, shared-memory computers like the RP3. I do not regret our decision to follow these beliefs uphill.

Acknowledgment

I would like to take this opportunity to thank the funding agencies that made the Ultracomputer research possible. Our initial support came from a DOE grant to NYU that predates my arrival (indeed this grant predates the DOE’s ‘‘arrival’’; it was first granted by the Atomic Energy Commission). In addition to continued support from the DOE, our project received several NSF grants. We also received significant ARPA support as a subcontractor to IBM’s RP3 grant. We thank IBM as well for committing to our support before they were assured of DARPA funding. We received two student grants from ARO and equipment grants from AMD, DEC, HP, DG, and IBM. Finally, we received a donation from Sphynx.

Retrospective:

Using Cache Memory to Reduce Processor-Memory Traffic

James R. Goodman

Computer Science Department
University of Wisconsin-Madison
goodman@cs.wisc.edu

While it has long been recognized that memory latency was a key parameter of performance, the impact of memory bandwidth (or its absence) has always been much harder to characterize. The complex relationship between latency and bandwidth is much better understood today than it was in 1982. Nevertheless, this relationship in the ever-varying context of "modern" system designs, created a fertile ground for studying a range of solutions to the same problem over many generations of computers: how to balance bandwidth and latency to provide a cost-effective, high-performance memory system [1,3].

This paper was an enthusiastic attempt by an assistant professor — who had never had a paper accepted to ISCA — to establish credentials in the area of single-board computer design, an exciting and growing market at the time. The paper reflects the exuberance and naivete of the era, where collecting data was as simple as figuring out how to use the trace mode on a VAX to generate a trace, and writing a cache simulator to evaluate the effects. With those simple tools, we studied everything from cold start/warm start phenomena to sector caches and block sizes, to replacement algorithms. Along the way, it became apparent to me that processors were approaching the point where it was preferable to allow them to sit idle rather than trying to keep them busy by switching tasks frequently. This seemed obvious to me because of the high-bandwidth, burst traffic required immediately following a task switch. While this conclusion led us down some interesting paths, apparently including the first publication of a snooping cache algorithm, we are still waiting expectantly for people to stop worrying about idle processors.

In the 1970s I was involved in the design of several add-on memory systems for IBM mainframe computers. Working on these designs, I had developed an appreciation for the design of cache memory, and the difficulty of supporting multiple processors in the process. In 1979 I had the opportunity to participate in the specification of the Intel 80286. Microprocessors were just arriving at the point where they were "interesting" to a computer architect, and I spent a lot of time thinking about Multibus-based systems. The Multibus standard allowed for multiple processors to share a bus, and to share commonly accessible memory on the bus but generally accessed memory on their own board. Microprocessors were just beginning to outrun DRAM memories. In discussions of how to support cache memory for the 286 I began to think about the implications of a small, on-chip cache, and how it could support a multiprocessor, and the idea came up of watching the bus from within the 286 chip. However, the multiprocessor systems I had studied were brute-force invalidation systems consisting of only two processors. Every write from a processor was conveyed to the cache of the other processor, where it was treated the same as an I/O operation, i.e., the remote cache was always checked, and invalidated on a hit. This resulted in a lot of traffic to the cache, since a check was required for every write.

In March of 1982, Carl Amdahl, on a visit to Madison, pointed out that write-back caches could reduce the invalidation traffic substantially: once a cache had been cleansed of a line, further invalidation requests were unnecessary if it could be guaranteed that the line didn't find its way back into the cache. This realization led us to a key reason that snooping is effective: the same mechanism can be used to detect a write — causing invalidation and exclusive access — and a subsequent read — causing intervention to prevent the reading of stale data.

The concept of a snooping cache developed over an extended period, so that when all the pieces finally fit together, I failed to see it as an important advance. In fairness, at least two other groups discovered the idea of snooping independently: Thacker and McCreight working on Dragon [4] at Xerox Parc and Steve Frank [2], architect of the Synapse N+1. In fact, when I wrote the paper, I believed that the important contribution of the paper was the recognition that caches — which usually had much higher bandwidth on the memory side than on the processor side—could actually reduce traffic from memory instead of increase it.

All of our studies were conducted in the context of a Multibus system, though we never actually implemented the idea at Wisconsin. (I was involved with the design of the Balance system developed in 1983 at Sequent.) We recognized that the addition of a couple of extra bus signals would make a protocol much simpler to implement, as well as provide better performance. Because of concern for commodity parts (the Multibus marketplace), we confined our design to one that could work in an unmodified Multibus system. From that experience I came to the realization that limitations imposed by commodity parts have their place, but that these limitations should not be allowed to stifle the search for novel solutions.

It was always apparent to me that there were better algorithms available if one relaxed our self-imposed constraint of Multibus compatibility. I was unprepared for the flood of papers introducing a plethora of variations. Many years passed before I was convinced that the seemingly small differences among these algorithms were important. I believe that, to this day, the most underrated contribution to understanding of snooping caches was Paul Sweazy's work on Futurebus, and his insistence that there had to be a common framework for comparing the algorithms. This work, of course, ultimately resulted in the MOESI model [5].

Finally, it's important to mention that my use of the term "we" in the original paper was not the royal we. As a junior professor I stumbled into an incredibly supportive and nurturing environment at the University of Wisconsin, one that allowed me to focus early on the research side of my career. I'm especially indebted to Jim Smith and David DeWitt, who led by example, and to Larry Landweber for his vision, leadership and advice. As acknowledged in the original paper, Phil Vitale and Tom Doyle participated in many stimulating discussions, both before and after this paper was written. Tswen-Hwey Yang built a powerful VAX trace tool that we used for this and much later work. She left the university before this paper was completed, and alas I have not heard from her since. David Patterson also played a critical role as a mentor during this period, and was one of the first to appreciate the importance of this work. Ed Davidson and Al Despain also provided critical advice and feedback during this difficult period.

I also want to put in a plug for supporting "small science" in the way that the National Science Foundation does so well. Support from NSF, particularly John Lehmann and, later, Zeke Zalcstein, has been critical in allowing many of us to develop our research programs. NSF support was enormously helpful to me, and this work resulted from my first NSF grant.

References

[1] D.C. Burger, A. Kägi, and J.R. Goodman, "Memory Bandwidth Limitations of Future Microprocessors," *23rd International Symposium on Computer Architecture (ISCA-23)*, May 1996.

[2] S. J. Frank, "Tightly coupled multiprocessor system speeds memory-access times," *Electronics*, Vol. 57, No. 1, (January 12, 1984), pp. 164-169.

[3] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Annual Symposium on Computer Architecture (ISCA-10)*, (June 1983), pp. 124-131.

[4] E. McCreight, "The Dragon computer system: an early overview." TR, Xerox Corp., Sept. 1984.

[5] P. Sweazy, A.J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proc. 13th Annual Symposium on Computer Architecture (ISCA-13)*, pp. 414-423, June 1986.

Retrospective:

Very Long Instruction Word Architectures and the ELI- 512

Joseph A. Fisher

Hewlett-Packard Laboratories
Cambridge, Massachusetts
jfisher@hpl.hp.com

VLIW Architectures and Region Scheduling

In this paper I introduced the term VLIW. VLIW was motivated by a compiler technique, and, for many readers, this paper was their introduction to "region scheduling" as well. I had put forward the first region scheduling algorithm, called Trace Scheduling, a few years before. Since region scheduling is a compiler technique, it is of interest to fewer people, but it enables superscalars and VLIWs with lots of instruction-level parallelism (ILP). Because I could see the power of region scheduling, I first began to think about VLIWs. I was fortunate in that this allowed me to coin the term Instruction-level parallelism, and to work out a lot of the original details and terminology of ILP, before many others believed it was important.

How VLIWs Came About

VLIWs came from my work as a graduate student at Courant. Ralph Grishman (my advisor) and I built PUMA, a CDC-6600 emulator that Ralph had designed (it worked well and eventually several were made and used to replace some big, old supercomputers of their day). One of my jobs was "chief tool builder" — I read the literature, learned the state of the art in ECAD algorithms, and then wrote tools to do most of the wire routing, partitioning, chip layout, simulation, and so on. You couldn't buy tools like that at a university then. The tools I wrote worked very well, but I was really frustrated with how hard it was to write, and especially maintain, the 64-bit horizontal microcode PUMA used (PUMA stood for "Processing Unit with Microprogrammed Arithmetic"). I originally thought of the problem that I solved with region scheduling as a hardware design problem. What I wanted to do was convert vertical (serial) microcode into horizontal microcode. The analogy to chip layout is clear:

- Chip layout involves converting a 1-dimension representation (a chip list) into a 2-dimensional representation (a placement); I was converting serial operations into 2- dimensional horizontal microcode

- Chip layout has nodes (chips) connected by edges (wire); I had operations connected by a data precedence relation.

- Chip layout tries to minimize wire length given the constraints of wiring; I had to minimize schedule length given the constraints of data precedence.

Because of this analogy, I was surprised when I realized that this was really a part of compiling, relying more upon compiler tools than CAD techniques. Fortunately, I was at Courant, which was then compiler heaven. (The holy scrolls were copies of Cocke & Schwartz, which included a catalog of optimizations; Ken Kennedy had just gotten his degree there.) The few people looking at this problem elsewhere were scheduling basic blocks, and then trying to iteratively improve the schedule by moving operations from block-to-block afterwards. The key insight was to recognize that this locked you to too many bad decisions, and you should instead look at a lot of code at once — for example a long execution trace. Trace Scheduling lets you do that, and frees you up to generate far more ILP. Since then, new region selection algorithms have suggested other regions of choice, often a more limited subset of a trace, reducing the complexity. Some of the region scheduling regimens that have gotten the most attention include Percolation, Superblock, Hyperblock and Trace-2.

Now that more and more ILP is present in microprocessors, region scheduling has become the technology of choice in high-end compilers.

Why Not VLIWs?

Given Trace Scheduling, I wondered why you couldn't build a RISC-style CPU with lots of ILP, and thus run really fast. Indeed, the farther my group and I went (I was at Yale by then), the more it seemed obvious that you could, and that it would be a good thing at least some of the time. I then learned a couple of things. First, you can get more people, a LOT more people, to come to your talk if you promise them bizarre sounding hardware instead of a compiler technique. Second, many people seemed to think that it would be very hard to build such a thing.

The first effect was good and bad. It's probably mostly my fault that a lot of people think of VLIWs as a weird kind of beast, rather than one of the natural alternatives once you start thinking about lots of ILP. I think that if I had presented these ideas in the context of a 2-issue compiled LIW, or a superscalar of any size, people would have thought more soberly about the idea. But because of region scheduling, I felt there was a real use for systems that could issue 7 (or many more) operations per cycle. This much ILP and weird long instructions to boot; it was too much for most people to accept. It was great for me professionally: I put forward a lunatic-seeming proposal, and then had it turn out to be practical, at least for some important uses, and not bad at worst in any case.

The second effect absolutely amazed me, and it still does. Why would this be impossible to build? I'd ask them, and I guess the lack of answers meant that you couldn't because no one ever had. (Really, in 1998 it seems amazing that people actually believed you couldn't build such a CPU at all.) Whether you'd WANT to build one seemed legitimately controversial and it still does, whatever I personally believe. The real question was: having built it, what now? Does this region scheduling stuff really work well enough to justify it? Answer: sometimes. Anyhow, I can't portray strongly enough the way in which this concept was greeted by the many minicomputer manufacturers John O'Donnell and I approached in trying to convince them to build one. I recently ran across a wonderful article in the San Francisco Chronicle (7/9/97, first Business Section page), quoting Ray Simar, Texas Instruments' program manager for their new generation of DSP chips:

The theoretical breakthrough came out of Yale University in the early 1980's. "I remember looking at the idea and saying these guys were nuts," Simar recalled. "I thought there was no way it would work in the real world." But three years ago, when Simar was given the job of coming up with a great leap forward in DSP, he revisited the Yale work with new appreciation. "At the end of the day what we thought was ridiculous was the best solution," he said.

It's just amazing to read someone being as forthright as that. Indeed, that is how people felt.

Some Naivete

There were a lot of aspects of this paper that seem naive from the perspective of 1998. Some of these were genuine naivete, others were simply artifacts of their time. The most significant is the lack of any mention of object-code compatibility as an issue for VLIWs. This issue was not on my radar screen. Being compatible with another company's binaries was an oddity, and most manufacturers changed architectures willingly. (If you copied someone else's, you sort of weren't even a legitimate computer vendor in some people's eyes, you were a "clone manufacturer".) People sometimes brought it up, but not often until Multiflow, when it was considered a big issue. (At Multiflow, we were "upward" compatible, but not as much as we'd have liked to be. You could run Trace 7 code on a Trace 14, but that was it. The 28, the one with 1024 bit instructions, you really needed to recompile for to run correctly.)

I never dreamed that there might someday be techniques that operated at run time that might solve the binary compatibility problem and change a lot of our other architectural considerations. (I've called these walk-time techniques, but nobody else seems to.)

A truly naive thing is that I described VLIW as an architecture, without really being conscious of the distinction between architectures and implementations. I didn't know to say it then, but VLIW is really a design philosophy, much like RISC, CISC, superscalar, vector processor, etc. To me, the part of VLIW that mattered then and matters now is the philosophy that one should get a lot of ILP in a processor without asking the hardware to do much to locate and schedule it. As with anything of this sort, there's a spectrum. I think of an implementation that expects operations to have been arranged so it can trivially put them in parallel at run-time as a good embodiment of this philosophy. I think of processors that significantly rearrange code, mapping architectural registers into physical registers, etc., and in general thinking instead of computing the answer, as embodiments of opposite. Does the object code actually have to have wide instructions in it for an implementation to be a VLIW? Not to me, and I'm not really interested in the question, any more than I want to know whether a particular processor is really RISC. I can tell the extent to which they follow this philosophy, and I think it's a good design philosophy.

At least two more truly naive things appear in this paper. First, I really ignored the whole question of exceptions. As anyone who builds real CPUs knows, you spend more time handling that problem than any other. Exceptions are a real pain for out-of-order processors. They're probably worse for superscalars than VLIWs, but no fun either way. Second, the memory system I thought was desirable, was, instead, baroque and silly. John Cocke tried to straighten me out. He suggsted that it would be enough to try to avoid references to the same bank, and that it would be a good idea to ignore this back-door nonsense. But I didn't listen.

Some Terminology

Finally, it's worth clarifying some terminology to put this paper in a more modern context. When John Ruttenberg and I laid out the basics of the high-level ELI architecture, we decided that register banks had to be split. We termed the combination of register banks and the functional units that took their operands from them a "cluster". That term has mostly stuck, but lately there have also been references to "split register bank architectures".

This paper addressed the problem of telling whether indirect references are to the same address or not, and called that problem "anti-aliasing". Because that term already had such strong meaning in the graphics world, l later renamed it "memory disambiguation", and that ugly term stuck. (Yale Patt always complains that his term, "the unknown memory address problem", was better. This shows again that short and ugly beats long and careful every time.) VLIW itself is another short and ugly term that stuck. I tried SPIE (for Static Parallel Instruction Execution). That turned out to be a conference name, which I found out when I used the term in a grant proposal and got on all the wrong mailing lists. I eventually came up with VLIW. I figured if VLSI could stick, why not VLIW? VLIW unfortunately emphasizes the long instruction implementation detail over the explicitly parallel instruction design philosophy that is really the key aspect.

One last thing I'd like to mention is that Mary Claire vanLeunen (author of the still wonderful "A Handbook for Scholars", revised ed., Oxford University Press, New York, 1992) taught me to write in the course of editing this paper, for which I'm still very grateful. Several others had tried; some had helped a lot, but this was where it took. "It's a lot like programming. Your goal is transparency," she told me. I figured that if she knew that about programming — not so many people knew that about programming then — she was undoubtedly right about writing. Recently, because of her lessons, I had reviews of a paper that said, "This is such a clearly written paper, it should be published on those grounds alone," and, from another reviewer, "This paper was written in an annoyingly juvenile style." Right!

Retrospective:

Characterization of Processor Performance in the VAX-11/780

Joel S. Emer

Digital Equipment
Shrewsbury, MA, 01545
Joel.Emer@Digital.com

Douglas W. Clark

Dept. of Computer Science
Princeton University, Princeton, NJ 08544
doug@cs.princeton.edu

In reminiscing with early VAX designers about the work in this paper, it has been difficult to recall how startlingly primitive our performance knowledge and approaches once were. While inside Digital and in the larger architecture research community we are now thoroughly indoctrinated in the quantitative approach to computer architecture and design, of which this paper is an early example, the situation in the early 1980's was quite different. In particular, while the VAX-11/780, which was introduced in 1978, was probably the preeminent timesharing machine on university campuses at that time, very little was known about how it worked or exactly what its performance was.

In particular, before 1980, even inside Digital the fact that some benchmarks ran at less than the widely-believed 1 MIPS was known to only a very small number of people. And the fact that on real multiuser workloads the 11/780 typically executed instructions at only 0.5 MIPS was apparently unknown. Furthermore, somewhat embarrassingly, both facts were unknown to the architects of some of the successor machines. That meant that those designs were optimizing to the presumed 5 average CPI of the 11/780, where in fact another 5 cycles per instruction were totally unaccounted for. It was only following some other measurements by one of us (Joel) in which a frequency counter was hooked up to record MIPS and he was shocked to read 0.5 MHz where he expected 1.0 MHz, that a more widespread account of the 0.5 MIPS rating was propagated. Still, so widely believed was the 1.0 MIPS number, in fact, that one of our ISCA referees didn't believe the data, making the "mandatory" recommendation to "explain why Table 8 and 1st bullet, pg. 23, seem to imply average VAX 780 instruction takes > 2 us; should be ~1 us."

In addition, while we have become accustomed to single chip microprocessors with minimal interfaces to probe their internal operations, the VAX 11/780 CPU spanned about 20 boards. One such board was the microcode store, which directed much of the behavior of the machine. That meant that one could probe the backplane of the machine to determine the address of each microinstruction executed. That's exactly what the measurement tool described in the paper was able to do. Furthermore, a microcoded processor like the 11/780 reveals a huge amount of detailed behavior this way, some of which was reported in the paper included in this volume.

While it would be nice to claim that all the work in the paper was premeditated as a comprehensive characterization of the 11/780, the microPC histogram tool used in the study was actually inspired by a single question that it wouldn't answer. It was probably late in 1980, and the company was in the early stages of the design of the VAX 8200, the first microprocessor VAX. Although it was a microprocessor, it wasn't on one chip, but the CPU core spanned three chips, not including the cache. Furthermore, the microcode had to be on an additional five chips. Since chip crossings were expensive, it was suggested that perhaps a two-level hierarchical microcode store would perform better. Thus, some small number of microinstructions could be included in the processor chip, and the remainder would live in the microcode chip. But with different latencies for different microinstructions, what would the performance be? The answer of course depended on the execution rate of each microinstruction. Unfortunately, we had little idea what the actual rates were.

The way to answer this question was obvious. Measurements of PC histograms for applications were commonplace, so why not measure the microPC histogram? Of course, as is invariably the case for questions that arise during a design, there was no time to conduct an extensive new study, especially one that involved building new measurement hardware. So a decision was made to build a single level control store for the 8200, as much for hardware complexity arguments as performance arguments. But the idea of a device to characterize microcode behavior was established, and given our preexisting belief in the value of accurate performance characterization, it was pursued in the expectation that next time we would be prepared with data to answer many other questions as they arose.

By the Fall of 1981, the first set of measurements was completed. Figure 1 shows the original graph that could answer the design question we were too late to answer. In addition, there was also a wealth of data on many of the arcane facets of use of the VAX architecture and the 11/780 implementation. At Dick Sites' suggestion, we created an annotated microcode listing that showed the relative execution count of each 11/780 microinstruction. Those listings were indispensable to several generations of VAX microcoders, who used them to determine the relative frequencies of different cases or typical microcode loop counts, and to budget microcode space. In addition, it was used to justify a variety of hardware/microcode tradeoffs.

Probably most significant was the two-dimensional instruction class versus operation cycle count table that appears as Table 8 in the paper. While such breakdowns of architectural and implementation statistics seems obvious and essential today, there were many novel applications at the time. This information was used to quell an internal attack on the (ultimately correct) performance claims of the VAX 8800. And the VAX 9000 architect carried around a marked up copy of the diagram with crossed-out entries, and updated values to justify the performance expectations of his machine. We believe that this data used in the early 80's was the most compelling evidence for performance claims that DEC designers had used to date, and was instrumental in establishing a firmly quantitative approach to performance inside the company.

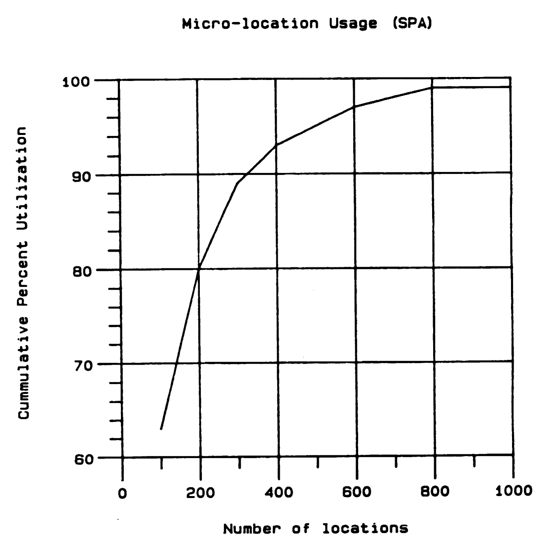
On the other hand, the most fun we had with the data happened in various design meetings, when, as seemed standard practice in those days, some clever designer would claim that a monumental performance gain could be achieved if only some cache were enlarged or the translation buffer were improved. It became a pleasant avocation to short-circuit these discussions with hard data, which inevitably showed that no single clever idea could cut VAX CPI significantly. Of course, this was not universally appreciated, as evidenced by the remark of a very senior designer, who in response to the interjection of a measured fact into a heated discussion, said, "Boy, you ruin all our fun — you have data."

As is often the case with industrial research, there was not a large incentive to publish results immediately, and this particular work was available within DEC for over two years before it was submitted to ISCA. There was also some understandable concern over the sensitivity of the data. In the end, we (and the corporate reviewers) decided that releasing the data was the right thing to do, as long as we didn't make the blunt observations that the 11/780 was only a 0.5 MIPS machine. Thus, noting that it was a little over 10 CPI and had a 200 ns. cycle time was okay, but no MIPs number was to appear.

In the end, we have been pleased by the acceptance of this paper as an example of the quantitative approach to computer architecture. We also have been pleased by the use of some other techniques exhibited by the paper, such as the separation of architectural and implementation statistics, the use of per-instruction metrics, and the use of better benchmark programs, especially those that include multiple users and system activity. It also seems clear to us in retrospect that this paper provided a service to designers of competitive machines (especially in academe), by quantitatively characterizing the most commonly used benchmark processor.

Both authors of this paper have continued to work on computer architecture. Joel has remained at Digital, and has worked on performance evaluation for a number of VAX processors. Doug was a designer of the VAX 8800 family, and worked on two further VAXs that never shipped. Both participated in the small corporate taskforce that led to the creation of the Alpha architecture. At that point their paths diverged, as Doug left Digital for an academic position at Princeton, while Joel has remained at Digital doing architectural research on various Alpha processors.

Figure 1. Original graph of micro-location usage



Retrospective:

A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories

Janak H. Patel

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801
patel@crhc.uiuc.edu

Origins of This Paper

The cache coherence solution proposed in this paper, now referred to as "Illinois Protocol", had its origin in my work on performance modeling and analysis of multiprocessors. Having completed the work on performance of multiprocessor interconnection networks in 1978 and first published in ISCA-1979 [1], I embarked upon extending the analysis to multiprocessors with cache memories. The analytical method used in [1] was readily extendible to more complex situations involving transactions between processors, caches, interconnection networks and memories. The interconnection network used was either crossbar or multi-stage Delta Network [1]. This analytical work I completed in Fall of 1980, and was later published in the *Transactions on Computers* [2].

The focus of these two papers was interference in the interconnection network with or without cache and therefore cache coherence did not get much attention, neither did bus based systems. However, things changed in 1981 when I came across a research project that Ed Davidson and his students were conducting at Illinois. Davidson had built a multiprocessor, called AMP-1 using the microprocessor Motorola 6800 and a synchronous bus. This system was designed around 1977-78 time frame by Bob Horst and Roy Kravitz and is described in an ISCA-1980 paper [3]. Several others were involved in performance modeling and measurement of the AMP-1, notably Joel Emer and David Yen. This is when I thought I could use my multiprocessor analytical method of [2] for single bus multiprocessor systems. While the AMP-1 did not have private cache, it still raised my interest in analyzing a bus based system with private caches.

In 1982, I was familiar with then prevalent microprocessor buses, namely Intel Multibus and Motorola VME Bus. I thought I should model a hypothetical multiprocessor system with caches and a bus like VME or Multibus. I was teaching a hardware lab that designed an interface for Multibus and as a result I was very cognizant of the lowest level details of its bus protocol. To model such a bus I needed to know various events that caused bus activities. A survey of literature found no bus based system with private caches. Most cache papers were directory based protocols and in addition the interconnection either was a crossbar or not mentioned. So I decided to just assume some arbitrary cache protocol. After all my goal was to provide a model and analytical method for such a system, not invent new cache protocols. As it happens with many innovations, they are often unplanned! So in Fall 1982, to make my analysis more realistic, I decided to define a cache protocol that had some practicality and low cost in relation to bus interfaces for Multibus. It was not very difficult to come up with a reasonable cache protocol. In that Fall of 1982, Marc Papamarcos started his M.S. Thesis under my supervision. He was very familiar with VME bus and Motorola microprocessors. So I asked him to work on a hardware implementation of this cache protocol for the Motorola VME bus. It so happens that the protocol is not directly implementable on either Multibus or VME bus without extending the capabilities of the bus in some way. However, we made sure that any modification to the bus were simple, practical and of low cost. Mark carried out a very detailed hardware design for the cache controller [4]. Mark also worked out details for implementing an indivisible read-modify-write for the Motorola 68K.

Many other researchers also were working on bus based cache coherence during that time period. When we had just finished writing the core of the paper we saw Jim Goodman's paper in ISCA-1983 [5]. We thought that since we found one paper on a bus based cache, we should include it in our performance analysis and we indeed did. This was not to prove the superiority of our protocol, it was merely a demonstration of analysis of different situations. The fact that our protocol did better, was just incidental. Since then a number of papers on coherence have appeared and indeed as I mentioned earlier, once you put your mind to it, it is not very difficult to come up with a variety of protocols. Nevertheless, Illinois protocol was different in several major respects: (1) Cache to Cache transfer of cache lines, (2) while Cache-to-Cache transfer takes place, the main memory also gets updated, and (3) all caches that have the requested line respond simultaneously, but a hardware priority network allows only one to perform the transaction.

I started presenting this work to industry considerably before it got presented in ISCA in June 1984. I gave talks at Texas Instruments, Intel, Digital, and IBM during 1983-84. In those days, questions often came about the difficulty of programming for cooperating processes on different processors in the presence of the cache. A common concern was that synchronization between parallel processes on different processors would be adversely affected by the cache protocol. Since this was such a common question, I started thinking of a simple explanation that could satisfy most casual programmers and engineers, very much like me.

Single Processor-Multiprogramming Paradigm

After considerable thought and analysis I was convinced that the cache protocol as defined with the indivisible read-modify write, was indeed so transparent to the programming synchronization that one could make the following statement with confidence. If your program with multiple threads can be run correctly on a single processor without any assumption on the order of execution among these threads, the same program will also run on a multiprocessor with private cache, where the threads are distributed in any number of processors. I started presenting this concept during my oral presentation in 1983-84. The importance of this paradigm is the sheer simplicity of it, which a non-expert in the intricacies of parallel programming semantics could easily understand and use during programming. It is good to contrast this simple paradigm with later paradigms with much more formal treatment and varying degree of serialization assumptions called weak ordering; see, for example, the paper by Adve and Hill in ISCA-1990.

In June of 1984, I presented this paper at ISCA in Ann Arbor, Michigan. Sitting in the front row was Dave Patterson, making copious notes in the proceedings. He asked the first question about my data showing the bus saturating at about 8 to 16 processors and stating that a company X had just announced a 32 processor single bus system. I also had performance data for zero coherence overhead, which still showed that for reasonable miss rates and bus transfer times, it was nearly impossible to put 32 processors on a bus without saturating the bus. The company X was defunct the next year. This only reinforced my belief in the need for good performance analysis in any architecture research.

IEEE Future Bus and Illinois Protocol

In November of 1984, I received a letter from Paul Borrill who was chairing the IEEE Futurebus Working Group. The letter stated that the working group had decided to include a cache protocol based upon our paper in ISCA-1984. He specifically stated that the solution that was being implemented was based upon our work and hence he needed my help in reviewing the standard. I asked my coauthor Mark to help them since he was located near where the Working Group was meeting. At a later date, the Working Group also took assistance from others, including Alan Jay Smith and Jim Goodman. As it happens with many standards, when the committee gets large the standard begins to cover everything under the Sun! This was no exception. The final Futurebus standard tried to accommodate a multitude of cache protocols, in the process they could not include the Illinois Protocol. The reason that Illinois Protocol could not be supported in Futurebus is that the IEEE bus did not permit the way the cache-to-cache data transfer were specified in the Illinois protocol. The irony of this is that this was precisely the reason that many others found the Illinois protocol attractive from a performance point of view. As a result, today the Illinois Protocol is found in several commercial multiprocessors, notably among them are SGI and Sequent. It is also found in some single microprocessors. To some it may not be readily apparent that a cache coherence protocol is also needed for single processors because of I/O.

Illinois protocol was also present in the early versions of the Nexgen microprocessors, which subsequently gave rise to the AMD-K6.

Final Thoughts

Over the years since the publication of Illinois Protocol, several in the community have told me that I should have patented this protocol. I do not believe this should be a goal of an academic researcher. Then, last year there was the news of the litigation between major computer corporations on patent infringement. One of the patents involved was on the topic of cache consistency protocols. The litigation involved large sums of money! Did I wish now that I had patented the Illinois protocol? The answer is still no. However, I do wish that companies honestly acknowledge academic research by referencing them in their literature. I am grateful that SGI and Sequent did acknowledge our work. ISCA proceedings have a wealth of novel ideas and if industry gives them due credit in public, costly intellectual property litigation could be avoided.

References

[1] J. H. Patel, "Processor-memory interconnections for multiprocessors," *Proc. 6th Annual Symp. on Computer Architecture*, pp. 168-177, April 1979.

[2] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Trans. on Computers*, vol. C-30, pp. 771-780, Oct. 1981.

[3] E. S. Davidson, "A multiple stream microprocessor prototype system: AMP-1," *Proc. 7th Annual Symp. on Computer Architecture*, pp. 9-16, May 1980.

[4] M. S. Papamarcos, *A low overhead coherence solution for bus-organized multiprocessors with private cache memories*, Technical Report CSG-29, University of Illinois at Urbana-Champaign, May 1984.

[5] J. R. Goodman, "Using cache-memory to reduce processor-memory traffic," *Proc. 10th Annual Symp. on Computer Architecture*, pp. 124-131, June 1983.

RETROSPECTIVE:

Implementing Precise Interrupts in Pipelined Processors

James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
jes@ece.wisc.edu

I became interested, reluctantly at first, in precise interrupts while part of the Cyber 180/990 project at Control Data in 1979. CDC had implemented imprecise interrupts in the 6600 and 7600. Precise interrupts could complicate an otherwise clean design, and weren't the kind of thing designers liked to think about. But the Cyber 180 was a virtual memory architecture, and the specification called for precise interrupts.

The earlier CDC STAR-100 had also been a virtual memory machine, and its schedule had suffered a substantial delay when recoverable interrupts had to be added to the design late in the day. The solution used in the STAR-100 and later Cyber 200 follow-ons was to implement an "invisible exchange package" — basically a dump of the pipeline contents that could be later restored.

With the STAR-100 lesson clearly in mind, my manager Jim Stockard asked me to assume responsibility for pulling together an overall precise interrupt strategy. Some pieces had been implemented, but there was no cohesive strategy. I believe Jim thought there was some connection between my fault-tolerant background and precise interrupts; I couldn't convince him otherwise. The onerousness of the task must have impressed me, because to this day I can clearly recall that specific meeting with Jim. Working on things like branch prediction seemed like a lot more fun.

Before I joined the project, some of the designers — including Denny Longnecker, John Pearson, and Terry Lyon, probably others — had worked out a method of restoring register state with a history buffer mechanism.

Mechanisms for handling memory stores and "multi-micrand" instructions remained to be worked out. "Micrands" were similar to RISC operations, and some of the complex Cyber 180 instructions required many micrands for execution. I started with the history buffer and added a method for handling multi-micrands by adding markers that delineated groups of micrands belonging to the same instruction. Stores were handled by signaling the store unit whenever all instructions preceding the store were known to be error-free; this was done via a simple reorder buffer-like mechanism.

Also at that time, I puzzled a little about other ways precise interrupts could be implemented, and decided that an alternative would be to replace the history file with a reorder buffer plus bypasses. But the history buffer method was in place, and the large number of bypasses seemed to be a problem.

I carried the precise interrupt problem back to Wisconsin and sat on it awhile — one of those projects worth doing at some point in the future. After Andy Pleszkun joined the faculty at Wisconsin, we started talking about the problem and started a joint research effort. At CDC, no serious study of the performance impact of precise interrupts was done — we felt it was small and were happy just to have something that worked. So, as part of our project, Andy and I measured performance impact of implementing precise interrupts (which turned out to be small).

Somewhere along the line, Andy and I came up with the future file method — it seemed to be the dual of the history buffer. Exactly how we came up with that one I don't recall. It seemed to round out the paper, though. Andy did all the performance work, and we spent many hours talking about the problem and the alternative solutions.

In retrospect, although neither of us thought it very significant at the time, the reorder buffer has probably turned out to be the main contribution of the paper. My colleague Guri Sohi made it really fly with the observation that the reorder buffer could be used for a lot more than supporting precise interrupts; it could also be used to support renaming and speculative execution. Sohi's method is described in another paper in this proceedings.

Retrospective:

HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality

Wen-mei W. Hwu

Computer & Systems Research Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801
hwu@crhc.uiuc.edu

Yale N. Patt

Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109
patt@eecs.umich.edu

"HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," was the first paper describing the HPS execution model that was published in ISCA. However, it was not the first paper published on the subject. The HPS paradigm was first presented at Micro-18 in two companion papers, the first introducing the execution model [1] and the second identifying the critical issues that needed to be dealt with if wide-issue, out-of-order execution, deep pipelined processors were to be viable [2]. The HPS paradigm was developed by three Berkeley Ph.D. students (Wen-mei Hwu, Mike Shebanow, and Steve Melvin) and their Ph.D. research advisor (Yale Patt). The idea crystallized during the summer of 1984 when Patt, Hwu and Shebanow spent the summer at Digital Equipment Corporation's Research group, at that time located in Hudson, MA.

We felt from the outset that high performance implementation required concurrency at all stages of the instruction pipeline. That meant wide-issue instruction supply which would only be viable if the subsequent execution of each of those instructions were decoupled from the set that each was fetched and decoded with. That meant dynamic scheduling of those instructions, and space for instructions awaiting execution to reside.

It also involved aggressive speculation to keep up the instruction supply, and an in-order retirement mechanism to handle precise exceptions. All parts of the mechanism are introduced in the first two papers.

HPS came about because the four of us believed that a microarchitecture that can take advantage of information available at run time can exploit concurrency inherently better than one that can't. The complexity of the microengine to do this was large, and the number of transistors required was also large. The concept also flew in the face of the popular paradigm of the day, RISC, which suggested that the compiler was the answer and that the hardware should be as simple as possible. In fact, many suggested that transistors should never be used to do anything that couldn't be accomplished by the compiler. This reached its extreme with the absence of a multiply instruction in the SPARC architecture — the multiply step was sufficient [3].

By 1985, the HPS research group had more than a year of experience studying the HPS paradigm in all its extended capabilities. An interesting question of the day, and the subject of this paper, was to design an HPS engine that one could reasonably expect to put on a single chip in the near term. The authors were fortunate to have at Berkeley at that time Professor David Hodges and his Ph.D. student, Greg Uvieghara. The critical storage structures on the chip, the Node Tables (aka reservation stations) and the Checkpointed Register Alias Table (used for handling precise exceptions) intrigued Uvieghara. Under the direction of Professor Hodges and with the consulting of Wen-mei Hwu, Uvieghara proceeded to investigate the viability of these storage structures for a current implementable design [4].

We limited the scope of all the structures in order to obtain a comprehensive HPS microengine that could be implemented on a single chip. We called the result HPSm, where "m" stands for the "minimal" implementation of the HPS paradigm. The chip supported three-wide issue, multiple functional units, the autocorrelation branch predictor, out-of-order execution, and checkpointed in-order retirement of instructions. In retrospect, we believe it was the in-order retirement of instructions, which provided the capability to handle precise exceptions, that subsequently made the HPS paradigm attractive for implementing commercial high-end microprocessors. Although most people did not embrace HPS when the original papers were published in 1985, insisting that (a) there were not enough transistors on the chip and (b) not enough parallelism in the code, a few industrial people were enthusiastic in their strong support. Particular acknowledgment goes to Lee Hoevel, formerly of NCR Corporation, and Fernando Colon Osorio, formerly of Digital Equipment Corporation, who believed in the concept from the outset. Today, 12 years later, almost every major high-end microprocessor embodies wide-issue, out-of-order execution, aggressive speculation, and in-order retirement.

Wen-mei Hwu received his Ph.D. from Berkeley in 1987, and joined the faculty of the University of Illinois, Urbana-Champaign, where he is now Professor of Electrical and Computer Engineering. At Illinois, he immediately focused his attention on a critical component of exploiting instruction level parallelism, the compiler. His IMPACT group has produced compiler technology that is vital to any high performance computing system today. Yale Patt joined the faculty of the University of Michigan, Ann Arbor, in 1988, where he continues to teach both freshmen and graduate students and direct Ph.D. research in high performance computer implementation. Mike Shebanow is the CTO and Vice President of HAL Computer Systems where he is responsible for the development of new high-end microprocessors. Steve Melvin is an independent consultant, operating mainly in the San Francisco Bay Area, but occasionally from his offices in Paris and Lisbon.

References

[1] Y.N. Patt, W.W. Hwu, and M.C. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Microprogramming Workshop*, Asilomar, CA, December 1985.

[2] Y.N. Patt, S.W. Melvin, W.W. Hwu, and M.C. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proc. 18th Microprogramming Workshop*, Asilomar, CA, December 1985.

[3] SPARC Programmer's Reference Manual.

[4] G. A. Uvieghara, W. W. Hwu, Y. Nakagome, D. K. Jeong, D. Lee, D. A. Hodges, and Y. N. Patt, "An Experimental Single-Chip Data Flow CPU," *Symposium on VLSI Circuits Design*, May 1990.

Retrospective:

A Retrospective on the Warp Machines

Thomas Gross *Monica Lam*

School of Computer Science
Carnegie Mellon University
Pittsburg, PA 15213

Departement Informatik
ETH Zürich
CH 8092 Zürich, Switzerland

Department of Computer Science
Stanford University
Stanford, CA 94305

Context and background

This paper presents the hardware and software architecture of the Warp machine, a parallel system of LIW processors. When the paper was written, CMU had just installed in a laboratory the first operational 10-cell system, which had been constructed by GE, one of the industrial partners. A second system was still under construction by Honeywell and was completed a few months later.

Developments after the paper

The Warp system was installed and almost immediately used for application development. Additional software demands were the price of user acceptance — our collaborators wanted to use the system over the network [3], they demanded optimized code [7] and a debugger. When the project started, the plan was to hand microcode a set of core vision library routines. When a compiler effort proved feasible, the application developers stated that programs would be "simple functions, about 1/2 a page of code". When the compiler was done, programs with a length of 10s of pages were written; the right hand side of one assignment alone contained 11,000 characters. (This statement had been generated by another tool. The compiler translated the statement correctly, but it took 30 minutes.)

The Warp array was connected to a UNIX workstation host — this organization contributed significantly to user acceptance of the Warp system. Now (in 1998) UNIX (or a variant) is the dominant operating system for supercomputers, but in 1985, connecting a compute engine (the Warp array) to a workstation was a novel implementation. Overall this decision was a solid one and set the tone for the next 10 years of system development.

1. Evolution

As we gained experience with the system, we noticed a number of features that limited the usefulness of the system. The biggest problem (already mentioned in the paper) is the tight coupling of the cells: if a cell sends data into a full input queue, then data are lost. This tight coupling made it impossible for the compiler to support the pipeline mode in practice.

Since the Warp machine at CMU performed really well for vision applications, several ARPA projects decided to use the Warp machine. The wire-wrap systems however were hard to replicate, and since at least one system was targeted for a moving platform, the users demanded an implementation on PC boards. CMU was in no position to produce systems, but GE or Honeywell (the partners of the wire-wrap phase) lacked the design expertise. Therefore the design team at CMU had a chance to revise the architecture, and we jumped at this opportunity. The most important chance affected the overall operation of the system: with added hardware flow control and a local controller, cells are no longer tightly coupled and can operate independently [1]. In addition, advances in VLSI allowed an increase in the sizes of the queues (4x), the program memory (4x), and the local memory (8x) — more memory always helps. The wire-wrap system was labeled the prototype, and the PC-board version was known as the Warp machine. The re-engineering of the Warp machine allowed us to put our compiler and application research results to immediate use, and although we worried a lot about engineering issues (like the cost, space, and power budgets), seeing the effect of our research on a real system was an exciting experience.

With these enhancements, the Warp machine became a solid platform for applications and research. The vision group developed a custom-program generator [6], and the increased flexibility made it possible to support other programming models, e.g., message passing [10]. Of course, Warp remained a good platform for systolic algorithms [9]. The Warp compiler pioneered software pipelining for LIW processors [7].

2. Applications

One Warp system was installed in the NavLab, a GM van converted by researchers in the Robotics Institute at Carnegie Mellon. The NavLab was a laboratory for research in autonomous road following, and a Warp machine (as well as a number of workstations) was installed in the van. The Warp machine turned out to be a good engine for neural net learning [8]; it also performed a variety of other vision and planing tasks. A Warp machine remained in the NavLab until 1989, when the van caught fire after the air conditioning system leaked liquid onto the computers. Warp machines were also used in other computer vision projects [4] (unfortunately often in environments that do not encourage publications), or for Kalman filtering [2].

In total, about 20 Warp systems were purchased by the government for various contractors (industrial research labs and universities) or government installations. The CMU team tried to convince GE to market the Warp machine to other research groups, but the GE unit was used to the "cost-plus" model of operation used in defense contracts: parts for the construction of a system are purchased after an order is booked. For the fast-moving computer industry, this mode of operation is deadly, and this experience with the defense industry motivated ARPA and us to search for a commercial partner for the successor project [5]. One machine was sold to a (friendly) foreign country, at least one attempt by an institution in another (neutral) foreign country to obtain a Warp died in the bureaucracy that controlled the export of high-technology. From a 1998 perspective, Warp performance may not appear impressive and it is hard to believe that this system was considered high-technology in its days. However, the reader should not be misled by the peak performance figure of 100 MFLOP/s. A few standard microprocessors connected together might have the same *peak* performance, but Warp delivered a high percentage of its peak performance [1], with impressive speedups over standard laboratory computers of its time. ([1] reports the performance of the Warp machine for various kernels and applications.)

3. Integrated Warp system

The Warp machine was highly successful, but it was not cheap ($350,000 w/o the workstation), and it was bulky. Furthermore, it could not support "industry-standard" languages like C or Fortran and it provided only basic support for message passing, with a fixed arrangement of channels (two channels in the left to right, one in the reverse direction). Soon after the paper was finished (and the design of the PC-board version was underway), we started to plan an integrated Warp system. This system became known as "iWarp" and was carried out in collaboration with Intel. [5] contains a detailed discussion of the shortcomings of the Warp system, as well as an evaluation of the iWarp design, implementation, and use for applications.

Looking back

When re-reading the paper, I was amazed to recall how much work the software group invested to deal with issues that we take today for granted. At the time the Warp machine was designed, address generation and loop control were moved to another unit, and the compiler worked hard to ensure that no cell ever read from an empty queue. Today, we take superscalar designs for granted.

The paper stated a number of conclusions, and most of them have remained valid till today: (i) Computer architecture is coupled to technology developments: when the Warp machine was designed, absence of hardware flow control and the arrangement of a linear array were a necessity. The PC-board version included hardware flow control and was able to support a much larger class of applications. A linear array is suitable for most applications, but once new technology (iWarp) allowed us to increase the number of nodes by (up to) 2 orders of magnitude, concerns about packaging, the shortest paths, and more general usage models led us to adopt a 2-dimensional torus. Subsequent systems, e.g., the Cray T3D and T3E, even moved to a 3-dimensional torus. (ii) Early identification of an *application* area is essential. Today we may want to add that continued collaboration with application experts is important to identify the strengths and weaknesses of an architecture; feedback by application experts pointed to interesting improvements. (iii) *Compilers are crucial* for an architecture project. At least this point, we hope, is nowadays universally accepted. (iv) *A balanced system* is essential if the architecture must host a variety of applications. The design of the external host turned out to be crucial: it allowed integration into a UNIX environment (which provided an environment accepted by the users) as well as high performance (data could be supplied to the Warp array so that it could operate at peak rate). This combination allowed Warp to win the "design competition" for the ARPA projects; other solutions were either based on a custom host (difficult to integrate into most environments) or lacked adequate I/O capabilities.

Systolic systems closely couple the computation units to the communication system — sending or receiving a word is as cheap as reading or writing local operands. The close coupling allows systolic systems (including Warp and iWarp) to support fine-grained computations (where each computation step requires a communication step). However, the Warp host interface exploits that many applications do not produce an output value for every computation step, i.e. the design recognizes the importance of coarse-grained computations.

The issue of coupling computation and communication remains an important one. Parallel systems still are difficult to program or achieve only low efficiency (or both). Warp demonstrated that a systolic system can be a cost-effective platform for many important applications. However an implementation based on PC boards is too expensive. An integrated solution is cheaper to manufacture but has higher setup costs — and as the cost for VLSI fabrication increased, the industrial partner of the integrated Warp project was not willing to continue the system. As of today, there exists no integrated system that combines good communication properties with high computation performance, e.g., for embedded systems in sophisticated cameras. Modern signal processors have closed the gap, but their communication performance is still not adequate in many situations.

The key idea of fine-grained coupling of communication and communication remains relevant. The Warp "family" of systems demonstrated for the first time the benefits of this idea for a programmable platform. Future architects, who may have access to other implementation technologies that combine implementation flexibility with low replication costs, will without doubt revisit fine-grained computations.

What happened to the authors

After the construction of the PC-version of the Warp machine, the Warp team members moved on to different projects:

Marco Annaratone was a faculty member at the time the paper was written; he joined afterwards the Department of Electrical Engineering of the Swiss Federal Institute of Technology (ETH) Zuerich. He is now director of the DEC Western Research Laboratory, Palo Alto, CA.

Emmanuel Arnould, a research engineer, returned to France.

Thomas Gross is still a faculty member at Carnegie Mellon and is still working on software.

H.T. Kung, the principal investigator of the Warp project started also the iWarp project and later moved to Harvard, where he is now the Gordon McKay Professor of Electrical Engineering and Computer Science.

Monica Lam was a graduate student in the Computer Science Department and is now a faculty member at Stanford University, where she directs the SUIF project.

Onat Menzilcioglu was a graduate student in the Electrical and Computer Engineering Department. A co-founder of FORE Systems, Inc., he served as President of FORE until January 1998.

Ken Sarocky was a research engineer; he left Carnegie Mellon to pursue a career in the computer industry in Japan and the U.S.

Jon Webb, a faculty member in the computer vision group, eventually left Carnegie Mellon to found Visual Interface, Inc., a company that develops hardware and software for shape photography.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu and J. A. Webb. The Warp Machine: Architecture, Implementation and Performance. I*EEE Trans. on Computers* C-36(12):1523-1538, Dec. 1987.

[2] R. S. Baheti, D. R. O'Hallaron and H. R. Itzkowitz. Mapping Extended Kalman Filters onto Linear Arrays. *IEEE Transactions on Automatic Control* 35(12):1310--1319, December 1990.

[3] B. Bruegge, Program Development for a Systolic Array. In *Proc. ACM SIGPLAN Symp. on Parallel Prog.*, pp. 31-41. ACM, New Haven, CT., July 1988.

[4] R. Dunley, Obstacle Avoidance Perception Processing for the Autonomous Land Vehicle. In *Robotics and Automation*, pages 912-918(Vol 2), 1988.

[5] T. Gross and O'Hallaron, D. *iWarp: Anatomy of a Parallel Computing System*. MIT Press, 1998.

[6] L. G. C Hamey, J. A. Webb and I. C. Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*, 48:246-264, 1989.

[7] M. S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1988.

[8] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky and H. T. Kung. Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. In *IEEE Second Intl. Conf. on Neural Networks*, pp. 143—150. July 1988.

[9] H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops*. Ph.D. thesis, Carnegie Mellon University, June 1990.

[10] P. S. Tseng. *A Systolic Array Parallelizing Compiler*. Kluwer, 1990.

Retrospective:

Memory Access Buffering in Multiprocessors

Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
dubois@paris.usc.edu

Christoph Scheurich

Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
christoph.e.scheurich@intel.com

At the time when the research work for the above paper was done, Michel Dubois was a starting assistant professor and Christoph Scheurich was a Ph.D. student at the University of Southern California. In this retrospective they both relate their experience on the work they did jointly more than 10 years ago.

The Advisor's Perspective

The problems of coherence and consistency have been addressed quite extensively in the past 10 years, to a point of information overload. It was not always that way. Back in 1982, the only definition of coherence was the one given by Censier and Feautrier in their landmark 1978 paper (the notion of "latest" copy) [1]. In that paper a central directory formed a bottleneck through which all cache requests had to go. It appeared to me back at that time that the definition was not sufficient in the context of a more distributed environment (even a bus-based system); moreover, as the speed gap between processor and memory widened, I realized that there would eventually be a time when memory accesses would have to be buffered in the processor to overlap them with each other and with computation.

I started thinking about these issues as I took a job in France in an R&D group in computer architecture from 1982 to 1984. During that time I dug out most of the literature that I could find on the topic. Mostly, the issues were touched lightly here and there in a few papers (clearly some people were thinking about the problems), and also there were Lamport's papers [2], totally ignored by the architecture community. If one of my colleagues from these early days in my career ever read this retrospective he will remember a presentation in which I explained in broad terms how to design a system such as the one shown in Figure 1, where each processor has a cache and an input and output memory access buffer.

Whether a multiprocessor system with caches and memory access buffers could ever work correctly in general was not at all clear, back in 1982.

When I took a position at U.S.C., I had the time to think more intently about these problems and wrote a paper for ISCA'85, which already included the definitions of strong and weak memory order. The paper was flatly rejected, which was extremely frustrating to me because I had the firm conviction that the material was important and fundamental.

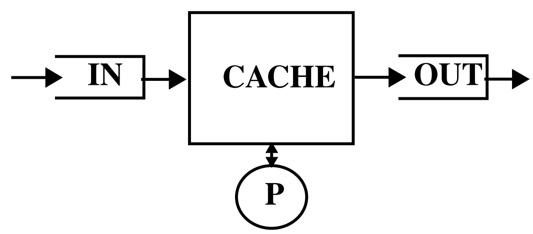


Figure 1

It appeared to me at that point that I needed help since thinking about the problem was consuming most of my time and energy and I had to teach at the same time. When I received a bit of money in the form of a research initiation grant from the National Science Foundation, I was able to start hiring students. I needed someone articulate, fearless, who has a strong sense of logic and common sense and I found that person in Christoph Scheurich who had been my teaching assistant for a semester.

After a learning period, we were able to have long technical discussions, which were very helpful. From that point progress was fast. I think we were both enthusiastic about the research. A new paper was submitted to ISCA'86. The reviews were mixed. I remember however a long, very technical review. Clearly we had connected with someone. Through the reviews we also learned of William Collier's work [3]. I remember being very relieved that someone else had been working so extensively on the topic.

Another paper was accepted the following year in ISCA'87 strictly on sequential consistency [4]. Obviously we still had problems with reviewers: I will never forget the comment of a reviewer who claimed that "he had polled his friends and no one cared about sequential consistency".

At this point more work needed to be done in refining the concepts and on performance evaluation. With another student and with the help of Faye Briggs who was then at Rice, we had developed a simple simulation environment [5]. However, the student left, and Faye Briggs went to Sun Microsystems. By that time Christoph deserved to complete his Ph.D. [6]. All my attempts at getting funding to continue the work failed. Through the vagaries of the funding process, I received some money to work on asynchronous algorithms. My focus shifted and I dropped the problem of coherence and consistency.

Meanwhile the work was still getting no recognition and again, this was extremely frustrating since I believed in the work and also there was clear interest from industry. It took until ISCA'90 before other groups published work on the same topic and finally our work started to be referenced [7][8].

The paper we published in 1986 was the first one to tie together coherence, sequential consistency and consistency relying on explicit synchronization. These issues were mostly ignored at the time, even by the large well-funded groups doing research on multiprocessors at other universities. The paper introduced the notion of general coherence (which does not rely on Censier and Feautrier's memory bottleneck) and of strong and weak memory access orders. Some of the terminology defined first in the paper such as "performed", "performed with respect to a processor", and "globally performed" has withstood the test of times and has become part of the vocabulary used to explain coherence and consistency.

The work had a large impact on both academia and industry. It ultimately led to innovations that were totally unexpected. Who could have predicted back in 1986 that architecture manuals would dedicate entire chapters to coherence and consistency and that the topic would generate the flood of ink that it did in the 1990's?

The Student's Perspective

The paper "Memory Access Buffering in Multiprocessors" was my first successful publication as a Ph.D student. However, after taking a class on data-flow computers, I was mostly interested in such architecture. It was a paper on "data-flow pipelines" that I had written for ISCA'86 that caused some unhappiness on the part of Michel Dubois (my academic advisor at USC) after he reviewed my first draft. He made some constructive suggestions but finished up by urging me to pick up on the problem of memory consistency that he had been working on before joining USC. I thought that the data-flow paper was pretty unique, I had no real understanding of what the research on memory consistency entailed, and had only mild interest in conventional multiprocessor architectures but decided that accord was in my best interest. The data-flow paper was of course summarily rejected by the ISCA referees and the research on memory consistency became the foundation of my Ph.D. thesis and also impacted other research activities outside of USC.

I read Michel's draft at least five times from the beginning to the end as well as the relevant papers by Censier and Feautrier and Lamport. When I started to understand the problem it became clear to me that thinking about parallelism involves some mental exercises that I could not perform in single steps - I needed a method to partition the problem. To that end, I spent some time with a text editor to come up with an early version of Definition 3.3 that is now part of the paper. Only later did I recognize this definition as significant. At the time I thought of it as a mental bridge.

Over a weekend I took some of Michel's earlier text and merged it with some of my own thoughts and embedded Definition 3.3. Then I went back to what I considered to be the real problem: data-flow pipelines. After Michel had some time to review my draft I expected that my stint in memory consistency research would come to a merciful end at our next scheduled meeting. I had no way of knowing that this meeting would have significant impact on my successful completion of the degree. I was thoroughly surprised when Michel's reaction to my write-up was very positive. Not only did he think that value had been added but it became obvious that he had spent some considerable amount of time thinking through the added concepts. Not surprisingly, I emerged from our first meeting on the new topic highly motivated.

In the following weeks we had many more meetings and it became clear that Michel was taking this work very seriously. We refined definitions, defined problems, and analyzed the conclusions. After we had both become fluent in the subject there were many long discussions on ordering scenarios that could become sometimes very obscure. Initially sequential consistency was our aim but then Michel shifted the focus to non-sequentially consistent models while maintaining correct operation (weak ordering). While sequential consistency made sense to us as it enforced the apparent sequential event ordering that we are all comfortable with, weak ordering initially appeared to me to be a formal definition for absolutely chaotic computer behavior. However, soon we managed to contain theoretically weakly ordered systems and defined rules that showed that such systems could work predictably. Because the paper is based on definitions there was very careful wording to be done as well. The simple performance models that demonstrated that weakly ordered systems can achieve higher performance than strongly ordered systems were added by Michel and made the draft ready for submission to ISCA.

The paper was accepted by the reviewers and the feedback was extremely helpful. However, one reviewer had rejected the paper outright without much elaboration. During the revision writing process we often wondered whether the negative reviewer had thought the topic mundane or whether he simply did not "get it." After some further feedback we finalized the paper hopeful that we had gotten the point across convincingly. The paper ends with the sentence: "We believe that more work is warranted in this direction." Whether or not researchers agreed with our other conclusions, that one certainly proved itself to be true. In the three years after the conference we published related research in various journals and conference proceedings [9] [10] and in 1989 my dissertation on the topic was accepted. The following year ISCA dedicated a session to the topic of multiprocessor memory models. By no means does the topic seem to be exhausted even today, as computer companies specify and build single processors and multiprocessors that vary significantly in their memory access ordering behavior.

What Happened Next?

Michel Dubois is currently a Professor in the Department of Electrical Engineering-Systems at the University of Southern California. He teaches courses on hardware design, computer architecture, parallel processing and performance evaluation. Although his mind still strays at times into the subtle intricacies of coherence and consistency his main research interests have drifted into system verification [11] and emulation of multiprocessors using FPGAs [12].

Christoph Scheurich joined Intel Corp. after receiving his Ph.D. at USC. At Intel he has worked on performance analysis, system architecture, and video capture implementations. Presently, he manages a group focused on real-time video imaging.

References

[1] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

[2] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690-691, September 1979.

[3] W. W. Collier, *Architectures for Systems of Parallel Processes*, Technical Report TR 00.3253, IBM Corporation, Poughkeepsie, NY, January 1984.

[4] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-based Multiprocessors," *14th Int. Symposium on Comp. Arch.*, June 1987, pp. 234-243.

[5] M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan, "Trace-driven Simulations of Parallel and Distributed Algorithms in Multiprocessors," *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, pp. 909-916.

[6] C. Scheurich, *Access Ordering and Coherence in Shared Memory Multiprocessors*, Ph.D. Thesis, Dept of EE-Systems, University of Southern California. Also Computer Engineering Technical Report No. CENG89-19, May 1989.

[7] S.V., Adve and M.D. Hill, "Weak Ordering--A New Definition," *Proc. 17th Int. Symp. on Computer Architecture*, pp. 2-14, 1990.

[8] K. Gharachorloo, et al., "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors," *Proc. of the 17th Int. Symp. on Computer Architecture*, pp.15-26, 1990.

[9] M. Dubois, and C. Scheurich, "Memory-Access Dependencies in Shared-memory Multiprocessors," *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990, pp. 660-673.

[10] C. Scheurich and M. Dubois, "Lockup-free Caches in High-Performance Multiprocessors," *Journal of Parallel and Distributed Computing*, 11, 25-36, January 1991.

[11] F. Pong and M. Dubois, "A New Approach for the Verification of Cache Coherence Protocols," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8, pp. 773-787, August 1995.

[12] L. Barroso, S. Iman, J. Jeong, K. Oner, K. Ramamurthy and M. Dubois, "RPM: A Rapid Prototyping Engine for Multiprocessor Systems," *IEEE Computer*, pp. 26-34, February 1995.

Retrospective:

Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors

Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
sohi@cs.wisc.edu

Background

Both Sriram and I started at the University of Wisconsin in the Fall of 1985; Sriram as a graduate student and I as an Assistant Professor. At that time, there was a lot of processor-related activity going on at Wisconsin: the Smith and Pleszkun paper on implementing precise interrupts [5] had just been published, and the PIPE project [3] was in full swing. I was looking for research topics to pursue. In graduate school, the papers that I had found the most fascinating were the IBM 360/91 papers. Even though I had written a Ph.D thesis about improving list-processing using associative structures, given the level of interest at Wisconsin in processor-related issues, it was natural to start working in that area.

Developing the idea

We started out in Spring 1986 pursuing two disparate lines of research. One was to simplify the logic needed to do out-of-order execution. Another was to develop a precise interrupt scheme that did not aggravate dependences as much as the reorder buffer (without bypass) mechanism proposed by Smith and Pleszkun (and did not have as much bypass logic as the reorder buffer with bypass). To carry out our studies, we had access to an excellent simulator of the Cray-1, developed by Jim Smith and Nick Pang.

The catalyst for our first goal was recent work by Weiss and Smith which studied Thornton's algorithm (aka scoreboard) and Tomasulo's algorithm within the context of the Cray-1, and proposed some variants of Tomasulo's algorithm [8]. Using that as a starting point, we set about looking for other ways to reduce the amount of hardware, especially tag matching logic, needed to implement out-of-order execution. The realization here was that not all the logic was performing useful operations; a lot of it was idle. By organizing the logic differently, the utilization could be improved. This led to the Tag Unit and the Reservation Stations and Tag Unit (RSTU) proposals in the paper. We then realized that the handling of tags could be simplified if instructions were completed in the same order as they were issued (this observation was also made in Tomasulo's original work).

In parallel with the above work, we were looking for ways of alleviating the aggravation in dependences caused by a reorder buffer (ROB). It occurred to us that these dependences, though aggravated, could be tolerated with an out-of-order issue scheme.

We realized that our two lines of thought were converging to a common point. We proposed and developed a common solution for both problems, the Register Update Unit (RUU), and started studying it, using a simulator which was developed by Sriram.

In the process of our experiments and studies, we learned several other things which we used to continue to refine our mechanisms. Perhaps the most important observation was that a machine of the form we were envisioning could be viewed as a reservoir with an inlet (instruction issue) and an outlet (updating registers, or instruction commit or retire in today's parlance) — the sustained rate of instruction flow at the outlet could be no greater than the sustained rate at the inlet. An important source of constrained instruction flow at the inlet was branches — instruction issue could not stop until a branch was resolved. Continuing instruction execution after a branch meant that we would need a way to separate machine resources corresponding to instructions after a branch from those before a branch, so that in case of an incorrect branch prediction, we could discard the incorrect work. We realized that the RUU mechanism could easily be extended to achieve this functionality — the extra information that was needed were pointers into the RUU which identified branches.

We continued to experiment with the simulator in the Spring and Summer of 1986. After ISCA'86, Jim Smith made us aware of the HPS work (see paper in this collection). My immediate reaction was one of disappointment — I thought that we had been "scooped". Later I determined that while we were solving similar problems, our solutions were quite different. I was especially encouraged later when Yale Patt told me that he was glad to see others "playing in the same sandbox".

What we learned and what followed

In the course of our experiments and studies, which took place both before the paper appeared as well as afterwards, we learned several things about the design of instruction-level parallel processors. This led to other papers which, at first glance, might appear to be unrelated to this paper. I describe some of these below.

Our first realization was the importance of maintaining the inflow to the "execution reservoir" — this is what is commonly referred to as the "front end" of the machine today. We realized the importance of instruction supply (branch prediction, instruction caches and branch resolution latency). The former two were not an issue in most of our studies since we used the Livermore loops as benchmarks; we studied the third aspect in an ISCA '88 paper [4]. We did not study these issues for other codes since we did not have simulators for a generic architecture until later.

We realized the (lack of) importance of data dependences vis a vis control dependences in an awkward way. As we were developing the (trace-driven) simulator, in an early version a bug resulted in data dependences not being enforced — instructions issued without regard for data dependences. When we fixed the error, the instruction issue rate did not decrease significantly, despite the somewhat long latencies of our underlying architecture — large buffers, coupled with the parallelism available in the codes we were studying, and the fact that much of the loss was due to branch resolution latency (instructions were not allowed to issue for some number of cycles after a branch) meant that data dependences were a secondary issue in such a machine.

Our next realization had to do with the flow of instructions in our "reservoir" (out-of-order core in today's parlance). We observed that restricting the flow of instructions in any of the execution pipelines could severely back up the machine. In particular, if we artificially throttled the memory pipeline, performance would suffer significantly. This led to our work on non-blocking caches. We initially did the non-blocking cache work on the Cray-1 simulator, using single instruction issue. Unfortunately, we were unable to show significant benefits due to a combination of single issue, low miss rates, and long functional unit latencies (and paper describing this work was rejected from ISCA). We redid the work for a MIPS architecture, using multiple issue, and were successful in getting our paper accepted to ASPLOS '91 [6].

We also learned that once memory requests (cache misses) are being overlapped, their latency becomes a secondary concern, given enough buffer space, and parallelism. Accordingly, we decided not to pursue cache enhancements that might provide minor improvements in miss rates (but were unlikely to improve the resulting instructions per cycle (IPC)) — we continue to use this observation to guide our research even today.

In our continued experiments with the RUU, we observed two other phenomenon, which led us to many other ideas. The first was that instructions close to each other in the RUU were rarely able to issue simultaneously because they were dependent — much of the simultaneous issue took place from points in the RUU that were far apart. The second was that when an instruction was committed from the RUU, most of the time its value was dead — another instruction in the RUU was going to overwrite the register. These phenomenon were perhaps artifacts of the limited number of primary registers in the Cray-1, and the poor quality of the compiled code. However, we later realized that these phenomenon were indeed more general phenomenon, which could be exploited.

The issue distribution phenomenon (along with other phenomenon that we had observed) provided one rationale for the Expandable Split Window (or Multiscalar) paradigm [1, 7]. This is discussed in the retrospective on "Multiscalar Processors". The register lifetime phenomenon also led us to carry out a detailed study of register traffic patterns [2]. This work, which was eventually published in MICRO-25 (after being rejected several times), provided a basis for our understanding of inter-operation communication behavior in a program, and serves as a rationale for the distributed register file design in a decentralized microarchitecture.

Looking Back

Today, most high-end microprocessors use speculative execution, out-of-order execution, and maintain precise interrupts. I would like to say that when we carried out the above work, we were confident in its eventual success. However, that was not the case. The late 1980s were a frustrating time for an academic researcher (especially an untenured one) to be working in the area of processor architecture. The strong anti-processor rhetoric in the research community, as well as the stream of paper rejections, caused us to doubt our own ideas and research philosophy on several occasions. For example, we realized the importance of binary compatibility (and the need to maintain the appearance of sequential execution), and the need to develop hardware solutions rather than resorting to software for solving most problems. However, at times we doubted this, and consequently doubted the solutions we developed. Were it not for the constant encouragement of my colleagues (Jim Goodman, Andy Pleszkun, and especially Jim Smith), it is likely that we would have given up in frustration and many of the ideas mentioned above would not have seen the light of day.

And I wish the spell checker had caught the mis-spelled word in the title (which I have left unchanged for this retrospective).

Acknowledgments

In addition to the people mentioned above, I would like to thank Larry Oliver and Zeke Zalcstein of the National Science Foundation, and the University of Wisconsin Graduate School who provided the funding for some of the research mentioned above. I would also like to thank Jim Smith and Sriram Vajapeyam for their comments on this writeup.

References

[1] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", *Proc. 19th Annual Symposium on Computer Architecture*, pp. 58-67, May 1992.

[2] M. Franklin and G. S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proc. MICRO-25*, December 1992.

[3] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schecter, H. C. Young, "PIPE: a Decoupled Architecture for VLSI," *Proc. 12th Annual Symposium on Computer Architecture*, pp. 20-27, June 1985.

[4] A. R. Pleszkun and G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors," *Proc. 15th Annual Symposium on Computer Architecture*, pp. 37-44, June 1988.

[5] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *Proc. 12th Annual Symposium on Computer Architecture*, pp. 36-44, June 1985.

[6] G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53-62, April 1991.

[7] G. S. Sohi, S. E. Breach and T. N. Vijaykumar, "Multiscalar Processors," *Proc. 22th Annual International Symposium on Computer Architecture*, pp. 414-425, June 1995.

[8] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 110-118, June 1984.

Retrospective:

The J-Machine

William J. Dally[1], Andrew Chang[1], Andrew Chien[2], Stuart Fiske[9], Waldemar Horwat[4], John Keen[9], Richard Lethin[5], Michael Noakes, Peter Nuth[6], Ellen Spertus[7], Deborah Wallach[8], D. Scott Wills[3]

[1] Computer Systems Laboratory, Stanford University

[2] Department of Computer Science, University of Illinois, Urbana-Champaign

[3] Department of Electrical and Computer Engineering, Georgia Institute of Technology

[4] Netscape Communications

[5] Equator Technologies Consulting

[6] Hewlett Packard Laboratories

[7] Department of Computer Science, Mills College

[8] DEC, Western Research Laboratory

[9] Silicon Graphics Computer Systems

Eleven years ago, at ISCA 14, we published a paper titled, "Architecture of a Message-Driven Processor" [1] marking the start of our J-Machine project at MIT. The project culminated with the construction of a working prototype in 1991 [2] and the evaluation of this prototype in 1992 [12, 15].

The J-Machine demonstrated the use of a *jelly-bean* part, a commodity part incorporating a processor, memory, and a fast communication interface, as a building block for computing systems. It was a *fine-grain* parallel computer designed to exploit large amounts of parallelism by balancing the use of silicon area between processor and memory. The J-Machine provided a small set of efficient communication and synchronization *mechanisms* that were used to support a broad range of programming models. It also provided fast user-to-user messaging without software intervention by having each message dispatch a message handler.

This retrospective reviews the history of the J-Machine project, discusses its contributions with the perspective of hindsight, and assesses what was learned from the project

Chronology

The chronology of the J-Machine project is summarized in Table 1. The project started at MIT in 1986. A team of students built a simulator for the machine, and we published a concept paper at ISCA in 1987. Funding to build the machine was secured in the summer of 1988. Intel joined us as an industrial partner and implementation started that August. We designed the machine using a very effective home-grown set of CAD tools [7] for

Table 1 Project Chronology

Date	Description
Sept 86	Project start
May 87	Initial architecture studies reported
Aug 88	Implementation started
Dec 90	Layout complete
Jun 91	First silicon
Jan 92	Second pass silicon
May 93	Evaluation reported

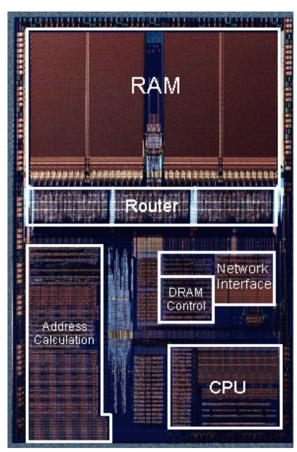


Figure 1: MDP Chip

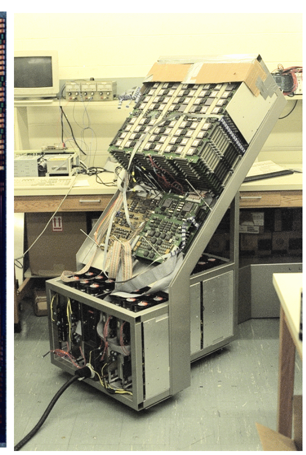


Figure 2: 1024-Node J-Machine

logic design, and Intel tools for physical design and verification. Layout was completed in December of 1990. The first chips arrived in June of 1991 and were running programs within a few hours. After revising the chips in early 1992 to correct a few bugs, we built three J-Machines: a 1024-node machine at MIT and 512-node machines at Caltech and Argonne. In parallel with the hardware development, several software systems were built for the machine [4, 19, 14, 9].

A die photo of the MDP chip with outlines identifying the major components is shown in Figure 1. The chip measures 1 x 1.5cm in a 1.2µm CMOS process. It was designed primarily using standard cells with hand placement for the data paths. The 32-bit integer CPU, at the lower right, measures 3.7 x 2.9mm.

Figure 2 shows a 1024-node J-Machine with the skins removed. Sixteen processor boards containing 64-nodes each occupy the top portion of the machine. The board stack communicates through elastomeric connectors [10]. The base of the machine contains an array of up to 80 disks. The peripheral bay is just below the processor stack. Peripheral interface cards developed for the machine include disk controllers, a distributed frame buffer for graphics, and two host interfaces.

Contributions

1. The Jellybean Part

The MDP chip was a prototype of our vision of a *jellybean* or commodity part: adding a processor and network hardware to a commodity memory chip. In the original paper we cited the advantages of high bandwidth, low-latency access to the on-chip memory and pointed out that the low latency access to the memory of other nodes in the network prevented the small amount of memory per node from limiting applications. The amount of memory reachable in a given number of cycles is important, not the amount of memory per node.

Our original plan was to build our own DRAM memory for the MDP prototype. Implementation constraints drove us to implement the actual MDP with 144Kbits of on-chip SRAM and 8Mbits of off-chip DRAM. This configuration simulated what we expected could be placed on-chip in the next generation (16Mbit) DRAM technology.

After demonstration of the prototype J-Machine in 1991, one of us (Dally) visited all of the major DRAM manufacturers to propose joint development of a commercial jellybean part by augmenting a 16Mbit DRAM with a 32-bit RISC processor, network interface, and router. There was no industrial interest in the project at the time.

Recently the idea of building jellybean parts combining a processor and memory, but without the network interface, has been revived [5, 13]. These projects are aimed at building stand-alone single-node systems, however, and do not address our original goal of developing building blocks for fine-grained parallel computer systems.

2. Fine-Grain Parallel Computing

Fine-grain machines, that balance processor and memory by silicon area rather than MIPS/Mbyte, achieve significantly better throughput per unit area, and per dollar, than do conventional coarse-grain machines [3]. This efficiency is achieved by having a larger fraction of *working* silicon and by reusing expensive memory more often. Efficient communication and synchronization mechanisms are needed to realize the potential of fine-grain machines, or these gains are swamped by overhead.

The bulk of the real cost (silicon area) in conventional computer systems is in memory. One can build a very competent 32-bit RISC processor (not including cache) in the area required by 100Kbytes of DRAM. Adding pipelined floating-point arithmetic raises this number to 500Kbytes. A machine with a simple pipelined floating-point processor and 256Mbytes of memory has only 0.2% of its silicon devoted to processing. This fraction of *working* silicon is decreasing with time as the memory in machines balanced by MIPS/Mbytes increases.

Conventional systems raise the fraction of working silicon by devoting large amounts of silicon to a single processor in an attempt to exploit instruction-level parallelism. This gives rapidly diminishing returns in performance per unit area. Doubling the area of a simple pipelined processor, for example, typically yields a performance improvement of less than 30%. The gains from a second doubling are significantly smaller. Increasing the number of processors, which gives nearly linear returns for some demanding programs, is far more efficient.

Consider an MDP built with today's technology incorporating a simple pipelined floating-point processor with a 64-Mbit DRAM. The fraction of working silicon on each of these modern jellybean chips would be 6%. A fine-grain machine built from 32 of these chips would have the same memory capacity and would cost about the same (same silicon area) as a 256Mbyte workstation with a more aggressive superscalar processor. The fine-grain machine has much higher memory bandwidth, lower local memory latency, and much greater peak performance. Even if our simple processor runs serial applications at half the speed of the more aggressive workstation processor, it outperforms the conventional processor by a factor of 16 on parallel applications and needs an average parallelism of only two to break even.

The efficiency of fine-grain computers depends on the availability of parallelism in applications. Our application studies on the J-Machine showed that there is plentiful parallelism in many applications. At small problem sizes, however, exploiting this parallelism requires short threads and frequent inter-thread interaction. The MDP's fast mechanisms enabled us to extract large amounts of parallelism even from applications run on small problem sizes. One string manipulation application, for example, gave a speedup of over 200 running a 1024-character string on 512 processors [12].

3. Mechanisms vs. Models

The J-Machine was designed with a set of fast primitive mechanisms for communication and synchronization intended to support a broad range of programming models. A message could be sent from user level with a single instruction, a message handler was dispatched by hardware on message arrival, synchronization was supported with presence bits on all memory locations and registers, and global (segmented) addressing was supported across the machine. These mechanisms were used to implement object-oriented [4], data flow [14], and message-passing [9] programming systems on the prototype hardware.

In 1987 when the MDP was proposed, people built machines specialized for one model of computation: data flow machines, shared memory machines, message-passing machines, and so on. It was generally accepted that a hardwired implementation of the programming model was required to achieve good performance. The J-Machine demonstrated that this was not the case.

The idea of building mechanisms in hardware and implementing programming models in software has received considerable attention [11][6]. However, two trends threaten this approach to parallel machine design. On the hardware side, the deep and complex on-chip memory hierarchy of modern microprocessors makes it difficult to build mechanisms without building a custom processor. The DEC Alpha 21164 processor used on the Cray T3E, for example, takes at least 20 cycles to wiggle a pin of the chip in response to a store instruction. It would not be difficult to add a path that bypasses the memory hierarchy and gives a much faster response.

On the software side, there is a disturbing trend toward applications that are written for the *least-common denominator* machine. Message-passing application are written in MPI or PVM and tuned to run on machines with 100µs message latency. Shared memory applications are written using a static process structure and tuned to operate with 10µs synchronization times. It is no wonder the people who write these applications conclude that they need large problem sizes to extract parallelism. Unfortunately, programs that are tuned to run on machines with slow mechanisms can't take advantage of fast mechanisms when they are available.

4. Fast Message Handling

In 1991, the J-Machine had the fastest interconnection network of any parallel machine in terms of start-up latency, throughput, and latency per hop [12]. Its performance was not surpassed until the announcement of the Cray T3D in October of 1993. The J-Machine's communication performance was due to a combination of a fast router, 3D packaging, and a streamlined network interface. Many ideas from the J-Machine network fabric have found their way into commercial machines including the Cray T3D and T3E and the Intel ASCI Red machine.

The network interface of the MDP, by providing a SEND instruction to transmit messages and hardware message dispatch on reception, set the standard for user-level messaging performance. Other experimental machines have taken a similar approach [11, 6, 17]. Software protocols have also evolved [16] to provide the same model of reactive messaging on stock hardware.

Lessons Learned

Fine-grain computing is feasible: Our experiments showed that it is feasible to extract parallelism with a thread size of a few hundred instructions from many applications and that with efficient mechanisms these applications can be efficiently executed on a machine with just 1Mbyte of memory per node.

Mechanisms work: We implemented several parallel programming models with performance competitive with hardwired implementations.

Building programming systems using the J-Machine mechanisms also taught us their shortcomings. For example, the streaming message SEND instruction of the J-Machine causes resource allocation problems. These shortcomings have been remedied in our follow-on projects [18].

There is no substitute to building it: By implementing the MDP we found that our initial estimates of cost and performance were in some cases far from the mark. The building process also fleshed out many challenging design and technology problems. Our work on high-speed signaling, for example, started from an observation that MDP performance was limited by pin bandwidth. The physical 1024-node prototype (too large to simulate) revealed challenging resource allocation and load balancing problems [8]. While it is certainly easier to quit after the simulation stage of a project, we found that the results at that stage lack reality and are often wrong.

Focus on the core issues: For the MDP the core issues were communication and synchronization mechanisms, fast context switching, and fine-grain thread and object handling. We were not studying instruction set design, pipelining techniques, circuit design, or logic design. We took a very conservative approach to these areas that sacrificed considerable absolute performance but greatly increased our probability of success. At various points in time we considered building our own DRAM, making extensive use of asynchronous logic, and pipelining the processor. In retrospect we are glad that we avoided this temptation of creeping featurism.

Why did we succeed? Intel's solid commitment as an industrial partner provided us with advanced CAD tools, manufacturing, and engineering talent. Their interest and the interest of students was held by a bold research agenda, rather than incremental measurement. The team expended as much time on validation as on design, meticulously cross checking the models at the instruction, RTL, gate and switch level; writing a comprehensive test suite; and simulating heavily for slow paths, hazards, and race conditions. We compromised some dimensions early, such as using standard-cell rather than full-custom design. Finally, the nature of the machine itself was amenable to building large systems: effort focused on the single-chip building block was multiplied by its ability to scale and form a large parallel machine.

A machine that requires new software is a hard sell: Technology transfer depends more on ease of insertion than on utility. Concepts from the J-Machine network, which are easy to insert, have seen the most use. The fast message interface, synchronization mechanisms, and fine-grain architecture have been largely ignored because they require major modifications to a microprocessor, completely new software, or both. The barrier to new ideas represented by existing software is formidable.

Incidental errors: We made a number of errors in the architecture of the MDP that were orthogonal to the issues of granularity and mechanisms. The machine had inadequate floating-point performance, too few registers, and did not operate the on-chip memory as a cache. These errors hindered our evaluation of the machine and were a barrier to getting applications programmers to target the prototype machine. Also, our software development would have been simplified if we had extended an existing instruction set rather than developing a new one.

Conclusion

The vision of a fine-grain parallel computer constructed from *jellybean* processor-memory-network components is even more compelling today than it was in 1987. As the fraction of working silicon in modern machines and the performance returns from aggressive superscalar implementations continue to decrease, explicitly parallel machines look even more attractive. Unfortunately, software compatibility remains a formidable barrier.

Using a fine-grain parallel computer as the memory of a conventional coarse-grain machine can lower the barrier of software compatibility. Such a *smart memory* machine can run existing programs unchanged, albeit with some increase in cost. Programs can then be parallelized incrementally, modifying critical loops and kernels one at a time to run on the fine-grain computer. By offering an incremental software path, smart memories make possible the transition from coarse-grained machines based on instruction-level parallelism to fine-grained machines based on explicit parallelism.

The J-Machine demonstrates the importance of building experimental computer systems. In the academic world, where we can afford to fail, we can build a machine based on unproven ideas. We also have the luxury of building a machine that demonstrates a vision of a computer system without concern for compatibility. It is essential to build and evaluate such machines. Simulation results do not reduce the perceived risk of new technologies sufficiently for industry to adopt them.

Acknowledgments

We thank all of the many people who contributed to the success of the J-Machine project. Special thanks go to Scott Furman on the MIT side and to Albert Yu, Greg Fyler and Roy Davison who made it all happen on the Intel side. We also thank Mark Pullen, John Toole, and Gil Weigand at ARPA for their vision and support.

References

[1] W. Dally, et al., "Architecture of a Message-Driven Processor," *ISCA-14*, pp. 189-196, 1987.

[2] W. Dally, et al., "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, pp. 23-39, April 1992.

[3] W. Dally, "A Universal Parallel Computer Architecture," *New Generation Computing*, pp. 227-249, 1993.

[4] W. Horwat, A. Chien, and W. Dally., "Experience with CST: Programming and Implementation," *PLDI*, pp. 101-109, 1989.

[5] P. Kogge, et al., "Combined DRAM and Logic Chip for Massively Parallel Systems," *Proc. 16th Conf. On Advanced Research in VLSI*, Computer Society Press, pp. 4-16, 1995.

[6] J. Kuskin, et al., "The Stanford FLASH Multiprocessor," *ISCA-21*, pp. 302-313, 1994.

[7] R. Lethin and W. Dally, "MDP Design Tools and Methods," *ICCD*, pp. 424-428, 1992.

[8] Lethin, Richard A., *Message Driven Dynamics*, Ph.D. thesis, MIT, March 1997. Also MIT/LCS/TR-721.

[9] D. Maskit and S. Taylor, "A Message-driven Programming System for Fine-grain Multicomputers," *Software - Practice and Experience.* 24(10), pp 953-980, October 1994.

[10] M. Noakes and W. Dally, "System Design of the J-Machine," *Advanced Research in VLSI*, pp. 179-194, 1990.

[11] R. Nikhil, G. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *ISCA-19*, pp. 156-167, 1992.

[12] M. Noakes, D. Wallach, and W. Dally, "The J-Machine Multicomputer: An Architectural Evaluation," *ISCA-20*, pp. 224-235, 1993.

[13] D. Patterson, et al., "A Case for Intelligent RAM," *IEEE Micro*, pp. 34-44, March/April 1997.

[14] E. Spertus and W. Dally, "Dataflow on a General-Purpose Parallel Computer," *ICPP*, pp. II231-II235, 1991.

[15] E. Spertus et al., "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," *ISCA-20*, 1993.

[16] T. von Eicken et.al, "Active Messages: A Mechanism for Integrated Communication and Computation," *ISCA-19*, pp. 256-266, 1992.

[17] Y.Kodama, et al., "The EM-X Parallel Computer: Architecture and Basic Performance", *ISCA-22*, pp.14-23, 1995.

[18] W. Lee, et al., "Efficient, Protected Message Interface in the MIT M-Machine", *IEEE Computer*, November 1998.

[19] D. Wallach, PHD: *A Hierarchical Cache Coherence Protocol*, S.M. Thesis, MIT, 1992.

Retrospective:

On the Inclusion Properties for Multi-Level Cache Hierarchies

Jean-Loup Baer

Computer Science & Engineering
University of Washington, Seattle, WA 98195
baer@cs.washington.edu

Wen-Hann Wang

Microcomputer Research Lab
Intel Corp., Hillsboro, OR 97124
wang@ichips.intel.com

When we wrote this paper, it had been over 20 years since caches had been introduced and they had become a regular feature in most computer systems. As applications were demanding more main memory and as the gap between processor speed and memory access time was widening (a familiar story today), the need for larger caches was clear. Like today though, increasing cache capacity was not a sufficient answer to the problem since as cache sizes grew, so did the cache access time. The need for a hierarchy of caches was apparent. Proposals for on-chip caches were burgeoning but, quite often, these caches were special-purpose: instruction caches only, top-of-stack caches etc. Shared second level caches for multiprocessor systems had also been proposed and even been implemented in one Japanese machine (Facom). However, the placement of second level caches in shared-bus multiprocessors, the parallel processing architecture of choice in the late 80's, and the impact of the cache hierarchy on snoopy coherence protocols had not been investigated thoroughly.

The conventional memory hierarchy of that time, i.e., cache, main memory, secondary memory was all inclusive: each component of the hierarchy was a subset of the component at the next higher level. It was easy to show that introducing two (or more) levels of caches without imposing some constraints on cache design (capacity, line size, associativity, replacement algorithm) would destroy this property, even in a uniprocessor context. While research on some special cases had been published, there had been no treatment of the problem in its full generality. One of the goals of our paper was to formally define rules that would enforce an inclusion property between the various levels of the memory hierarchy. This MultiLevel Inclusion (MLI) property was to hold for a tree-like hierarchy so that caches at a given level could be shared by lower level caches as could be needed in multiprocessors. A second goal was to explore the design space of cache hierarchies in shared-bus multiprocessor structures and to define protocols so that MLI was enforced for these architectures.

One pleasant memory about writing this paper is that we had fun proving new ideas and exploring new cache structures without having recourse to time consuming simulations! Of course, we did the simulations later, when we showed that MLI was instrumental in reducing coherence checks in a shared-bus multiprocessor system [1] or for solving synonym problems for virtually addressed caches [2] or for fast cache simulations [3].

Until quite recently, MLI was a given in the implementation of cache hierarchies for single processors or multiprocessors with a small number of processors like in SMP clusters. However, there are specific instances now where MLI might not be warranted. The first one is when the two lowest levels of the hierarchy are on-chip, or when the second level cache L2 is "glued" to the processor chip like in the Pentium Pro. There even have been proposals to have L1 and L2 be mutually exclusive [4] in order to maximize the amount of information kept on-chip. Nonetheless, MLI still holds between the on-board cache (L3) and L1 and L2 taken as a whole. The second example is when in a cluster an L2 (or L3) remote cache is shared by the processor caches of the cluster. This remote cache, as e.g., in the Sequent CC-NUMA [5] can be such that it contains only data that is not homed in the cluster. However, MLI is enforced for the remote data. And thirdly, an excessive amount of associativity at the L2 level might be needed to enforce "pure" MLI. This pressure on the associativity component was what led us to refine a "partial inclusion" protocol that had originally been proposed by Wilson [6].

Two of the three shared-bus multiprocessor structures that we described, namely those corresponding to Multi's (as defined by Gordon Bell) and clusters with a partial inclusion protocol similar to the one we sketched in this paper, are very popular. The third one, a multiport cache hierarchy, was included because it was the architecture of an existing machine. To our knowledge, it is not a structure implemented in any recent multiprocessor system.

As a final comment, on the light side, we had used the notation L1, L2 etc. for the various levels of caches in our submitted manuscript. One of the referees was adamant that we changed the notation to C1, C2 etc. Unfortunately, we complied! It is clear that we should have not done so. The morale of the story is that authors should not always listen to referees.

About the authors

After completing his Ph.D. at the University of Washington on "Multilevel cache Hierarchies," Wen-Hann Wang joined IBM Watson Research center in 1989. Since 1991 he has been with Intel Corporation where he worked on P6 platform architecture development for a few years. Currently he is a Principal Researcher at Intel Micro-Computer Research Labs where he leads a number of system research activities.

Jean-Loup Baer is still a Professor of Computer Science & Engineering at the University of Washington. The design and performance of memory hierarchies remain his primary research interest. He gratefully acknowledges the constant support from the National Science Foundation, not only for this paper but also throughout his academic career.

References

[1] Jean-Loup Baer and Wen-Hann Wang, "Multi-level Cache Hierarchies: Organizations, Protocols and Performance", J*ournal of Parallel and Distributed Computing*, vol. 3, pp. 451-476, 1989

[2] Wen-Hann Wang, Jean-Loup Baer and Henry Levy, "Organization and performance of a virtual-real cache hierarchy", *Proc. of 16th Int. Symp. on Computer Architecture*, pp. 140-148, 1989

[3] Wen-Hann Wang and Jean-Loup Baer, "Efficient trace-driven simulation methods for cache performance analysis", *ACM TOCS*, vol. 9. no. 3, pp. 222-241, 1991

[4] Norm Jouppi and Steven Wilton, "Tradeoffs in Two-Level On-chip caching", *Proc. of 21st Int. Symp. on Computer Architecture*, pp. 34-57, 1994.

[5] Tom Lovett and Russell Clapp, "STiNG: A CC-NUMA computer system for the commercial marketplace", *Proc. of the 23rd Int. Symp. on Computer Architecture*, pp. 308-317, 1996

[6] Andrew Wilson, "Hierarchical cache/bus architecture for shared memory multiprocessors", *Proc. of 14th Int. Symp. on Computer Architecture*, pp. 244-252, 1987

Retrospective:

Evaluation of Directory Schemes for Cache Coherence

John Hennessy

Computer Systems Laboratory
Stanford University
jlh@mojave.stanford.edu

Origins of this paper

This paper was born from work done in 1987 that aimed at exploring ways to build scalable shared-memory multiprocessors. This research began in a special graduate seminar class in Spring of 1987, which we had initiated to brainstorm on architectures for shared- memory multiprocessors. Prior to beginning the work in this paper, we considered a variety of schemes for avoiding cache coherence.

In particular, we initially explored software-based cache coherence schemes. We concluded that while such schemes might work well for highly structured, loop-intensive scientific applications, software-based cache coherence was too inefficient for less structured scientific applications as well as for operating systems software. For such software environments, we concluded that either too many cache invalidations would be needed, or too much data would be need to be kept uncached.

After concluding that software-based coherence would not be sufficient for a broad range of applications, we still felt that a coherence scheme that was fully general might be too expensive or hard to scale, so we began exploring a variety of schemes that significantly limited sharing. One of the first thoughts we had was to consider a scheme that allowed only a single reader for a shared writable data item. This scheme generated part of a basis for this paper, as well as for a broader exploration of directory approaches.

Developing the Directory Idea

In trying to avoid the broadcast nature of snoopy-based coherence, we decided to try using directory-based coherence as a possible approach. At the time of this paper, our focus was driven by one belief and one limitation of our experimental data. Our belief was that any alternative to snooping should be competitive with snooping, even at small processor counts. Since earlier papers on snooping (especially Goodman's landmark paper included in this collection) had raised questions about the performance implications of the restricted bandwidth of a centralized directory, we felt that it was important to carefully evaluate the bandwidth requirements for a directory-based scheme.

The limit in our investigations at that time was that the shared-memory multiprocessor traces that we had available were for small processor counts. Thus, our initial research focus, and the major focus of this paper became to evaluate the performance of different directory schemes both within themselves and compared to snoopy-based coherence at small processor counts.

We believed it was important to create a framework for directory schemes that would include the earlier work of Tang, Censier and Feautrier, Yen and Fu, and Archibald and Baer as well as the limited sharing schemes we were interested in. This framework, which labels a scheme as Dir_nB or Dir_nNB, where n refers to the number of directory entries per block of data and B or NB describes whether the system uses broadcast when the number of sharers exceeds n or whether the system simply invalidates one of the sharers to make room for a new one. This framework captured one scheme that we found attractive from an implementation viewpoint: Dir_1NB and more generally described a space of possible directory approaches.

We focused our evaluation on four protocols: two directory-based protocols (Dir_1NB and Dir_0NB, the latter of which is the Archibald and Baer scheme) and two snoopy protocols (a write-through invalidate scheme and the Dragon write-update protocol). Later in the paper, we added an analysis that estimated the performance of the Berkeley Ownership protocol. We focused our evaluation by looking at bus cycles as the primary metric of protocol performance. Given the four-processor memory traces and our interest in seeing if directory schemes were competitive with snoopy schemes, this was a local choice.

The key insight from the performance studies in this paper was that directory schemes are reasonably competitive with snoopy schemes, especially if synchronization traffic (especially spin locks) are removed. (The use of such spin-locks heavily penalize the Dir_1NB schemes.) It was these results that gave us confidence that directory schemes might be efficient enough to compete with snoopy schemes, while providing the possibility of scaling to larger processor counts. This led to some of the most interesting and important speculations, which are the subject of the last two sections of the original paper.

Looking Forward: Scalability of Directory Schemes

Section 6 of the original paper investigates both the scheme that we initially implemented in DASH (Dir_nNB) as well as Dir_iB, a scheme most similar to that implemented in the HaL S1 machine. We also speculated on the possibility of using coarse representations of multiple caches, which is the scheme used in the SGI Origin for more than 128 processors. The competitive performance of Dir_nNB, even in a bus-based environment, which made broadcast as cheap in bus cycles as a single invalidate, motivated us to explore directory approaches further. We reasoned that the phenomena of having small numbers of sharers for a data item that was written with some reasonable frequency would probably hold when we looked at traces with larger number of processors. (Of course, this property depends on the separation of synchronization traffic, which has very different behavior.)

Perhaps the most important insight in this paper is contained in the first paragraph of the conclusions: "The basic bandwidth limitation to the memory and the directory can be mitigated by distributing them on the processor boards. This technique allows the bandwidth to both the memory and the directory to scale with the number of processors." This key observation that directory bandwidth and memory bandwidth could be scaled by distributing both was the key insight that led from this paper to the DASH project. This paper enabled us to see that such directory schemes would likely have acceptable performance and thus, encouraged us to explore distributed directory cache-coherence. As we say in the last sentence of the paper: "If this data holds for large-scale multiprocessors, directories will provide an efficient method of implementing shared memory."

Looking back: what we knew, what we guessed, and what we could not guess

Perhaps in retrospect, our concern with the competitiveness of directory schemes compared to snoopy schemes was misplaced. The key issues for scalable cache-coherent shared-memory have clearly turned out to be the scalability of the coherence scheme, the implementation complexity and overhead of the coherence protocol, and the performance of such protocols at larger processor counts. Nonetheless, this paper was a vital link in the chain leading to further investigation of distributed directory protocols. This paper gave us the confidence that directory schemes could be competitive with snoopy schemes at small machine sizes. We knew that bandwidth of directory schemes could be scaled by distributing the directory with the memory. Finally, we reasoned that directories would yield good performance except under pathological sharing situations—situations where the communication rates between processors would probably lead to unacceptable performance for any multiprocessor architecture. What we clearly did not understand were the challenges in implementing distributed directory coherence. Overcoming that challenge is the story of the DASH project and another paper in this collection.

Acknowledgments

A number of people helped make this paper what it became. DARPA supported our research efforts. John Toole and later Gil Weigand were the DARPA program managers. We appreciate their encouragement to strive for the big payoff. Anoop Gupta also played a key role in urging us to consider scalable shared-memory and later helped lead the DASH project.

RETROSPECTIVE:

Weak Ordering—A New Definition

Sarita V. Adve

Dept. of Electrical & Computer Engineering
Rice University
Houston, TX 77005 USA
sarita@ece.rice.edu

Mark D. Hill

Computer Sciences Dept.
University of Wisconsin-Madison
Madison, WI 53705 USA
markhill@cs.wisc.edu

Introduction

We began work on "*Weak Ordering—A New Definition*" [3] in early 1989 while Sarita Adve was a first-year graduate student and Mark Hill a second-year assistant professor at Wisconsin. It now seems obvious that an interface for shared-memory must be defined. It also seems obvious that such an interface must consider interactions among reads and writes to all shared-memory locations, and must not refer to hardware structures such as caches and write buffers. In early 1989, however, most work related to shared-memory semantics was on cache coherence. Such work reasoned about interactions between reads and writes to a given cache line in isolation, focusing on hardware protocols to ensure that the effect of a newly written value eventually propagated to all processor caches. Only a few papers had been written about a more comprehensive model of memory [8, 9, and references in the main paper].

Our work was primarily motivated by the pioneering work on weak ordering by Dubois, Scheurich, and Briggs [5]. The motivation and intuition behind weak ordering were compelling. However, as originally defined, weak ordering had two problems: (1) the definition was hardware-centric and did not seem to be appropriate as a programming model, and (2) the definition appeared to unnecessarily constrain hardware. These observations steered us towards the following two questions:

- What are the *minimal* conditions that a shared-memory model must impose on hardware?
- How could the shared-memory model be best presented to programmers?

For a while, we viewed the above two questions somewhat independently. The defining moment of this work was when we realized the connection between the two questions and redefined the memory model to be a contract between hardware and software. Specifically, we saw a weakly ordered system as one that provided Lamport's sequential consistency [13] to data-race-free programs. Our model was subsequently dubbed *data-race-free (DRF)* or *data-race-free-0 (DRF0)*.

We next describe the process that led to the paper and briefly summarize later work in the area. Release consistency and the notion of properly labeled programs were developed concurrently with our work and are based on similar ideas [7].

The Process

Search for a weaker model for hardware

Our initial work was hardware and performance-centric, and focused on defining a set of conditions that were less constraining for hardware than Dubois et al.'s weak ordering. We would consider common application characteristics, and develop hardware constraints that would give "reasonable behavior" for those (informally characterized) applications. In this process, we defined multiple models that relaxed consistency requirements in different ways at different points in the program (e.g., at the acquire or release of a semaphore). These models, although less constrained than Dubois et al.'s weak ordering, were nevertheless similar in style to the definition of weak ordering, and suffered from the drawbacks we sought to alleviate.

A characterization of software

Our first key departure from Dubois et al.'s work was to use partial orders instead of real time in our specifications. Using any notion of real time made the specifications harder to understand from the programmer's viewpoint and unnecessarily constrained hardware. The use of partial orders was motivated largely by the work of Lamport (e.g., [12]) and of Rob Netzer and Bart Miller [14], our colleagues at Wisconsin.

The second important step, in Summer 1989, came from making a deeper connection with the work by Netzer and Miller [14]. They were working on detecting data races in a program, and used a variant of Lamport's happened-before partial order relation for formalizing the notion of a data race. We realized the connection between their characterization and the application behavior that we were attempting to characterize for our weak models. It became clear (at least intuitively) that weak ordering and the weaker models we were trying to develop appeared sequentially consistent to data-race-free programs.

We now had a formal understanding of what was needed from the application. We were still, however, grappling with hardware conditions for our new ideal memory model. Nevertheless, at this point, we thought we had a well-defined path that would lead us to the ideal model. All we needed was to determine the minimal set of hardware constraints that would provide sequentially consistent results for data-race-free programs, and call those constraints our new memory model (or so we thought).

Minimal conditions for the hardware and model

For almost three months, we frequently invented a new "model of the day." We would formalize a set of conditions that appeared necessary and sufficient, but soon would discover another way to weaken those conditions. To prove or disprove the correctness of our conditions, we made use of the formal methods developed by Shasha and Snir [16] as well as ad hoc techniques. In late October 1989, we realized that not only were the absolutely minimal hardware constraints elusive, but also that a model defined in terms of the type of constraints we were proposing would be quite complicated.

At this point, we realized that we needed to move beyond viewing the model as purely a set of hardware constraints. The defining moment of this work came with the observation that weak models could be viewed simply as a contract between hardware and software. Given that we had already defined a set of conditions for software, the only necessary condition for hardware was to appear sequentially consistent for the proposed software. Further, we could develop different models by determining different software conditions; the hardware for those models would simply need to appear sequentially consistent to the specified software.

Subsequent Work

After the 1990 paper, most of our immediate work focused on formalizing the software conditions for which commonly used system optimizations would not violate sequential consistency, and on formulating further system relaxations that would not violate sequential consistency. Some of this work was joint with Kourosh Gharachorloo, Anoop Gupta, and John Hennessy of Stanford. A common theme throughout this work was that most problems at first appeared to have deceptively simple solutions; however, formally proving the correctness of the solutions proved to be quite difficult. Our eventual framework to do these proofs benefited immensely from previous formal work by Collier [4] and by Shasha and Snir [16].

The flexibility afforded by defining a memory model in the new programmer-centric way is arguably most evident in the software shared-memory work that followed later. Lazy Release Consistency [11], arguably the most widely cited algorithm for software shared-memory, is weaker than release consistency. However, both release consistency and lazy release consistency obey the data-race-free model since they both appear sequentially consistent for data-race-free programs. Thus, for programmers who write data-race-free programs, these systems are equivalent.

Over the last few years, a rich body of literature in the area of memory consistency models has developed. This includes new models for hardware and software shared-memory, performance evaluations, theoretical frameworks for formal specifications and proofs, and highly successful methods to reduce the hardware performance gap between consistency models. Most advances, however, have been in the domain of hardware and runtime systems. The performance impact of relaxed consistency models on compiler optimizations is still unclear. Programming languages and environments have also only recently begun to address the issue more explicitly, with many supporting relaxed models (e.g., Java, OpenMP, and POSIX). A tutorial on the subject and an overview of recent advances appear in [2,1].

Although memory consistency models are now well-understood, there is no consensus yet about the best consistency model. At the time of this writing, commercial multiprocessors supporting sequential consistency and relaxed consistency models are available. The Digital Alpha and IBM PowerPC processor architectures support relaxed models similar to DRF or release consistency, Intel IA-32 and current SPARC processors support derivatives of a relaxed model called processor consistency, while processors from HP and MIPS support sequential consistency. Recent hardware optimizations that reduce the hardware performance gap between various consistency models [6,15], the lack of quantitative data on the benefits of relaxed models for compiler optimizations, an absence of widely used programming standards for shared-memory, and the requirement on vendors to keep their systems backward compatible are some of the factors that have made a consensus difficult. One of us (Mark Hill) has recently used some of these factors to make an argument for returning the hardware/software interface to sequential consistency [10].

Summary

In summary, the key to our work was (1) we took a more programmer-centric view of the problem compared to the more prevalent hardware-centric view at that time, and (2) our persistence in seeking the minimal possible constraints for the hardware interface. This resulted in our redefining the problem in programmer-centric terms, enabling a better understanding of some of the fundamental issues. It is perhaps worth noting that when we began this work, the problem seemed deceptively simple, and a highly respected senior colleague actually warned us that we were getting into what appeared to be a closed area!

Acknowledgments

We thank those acknowledged in the original paper and Jim Goodman, whom we regrettably omitted. We thank the sponsors of this research for the confidence they showed in a new professor: Michael Foster and Zeke Zalcstein of the National Science Foundation, Rae McLellan of Bell Labs, Chuck Thacker then of DEC, Doug Johnson of TI, Chris Hsiung then of Cray, and the University of Wisconsin Graduate School. We want to especially note the acceleration brought to our careers by the NSF's Presidential Young Investigator program.

Biographies

Sarita V. Adve continued work on memory consistency models for her Ph.D. thesis, under the supervision of Mark Hill and supported by an IBM graduate fellowship. She joined Rice University as an assistant professor in 1993, where she has worked on techniques to improve and evaluate the performance of shared-memory systems. She received an NSF CAREER award in 1995, an IBM Partnership award in 1997, and an Alfred P. Sloan research fellowship in 1998.

Mark D. Hill continued research on memory consistency models, large caches, translation lookaside buffers, and page tables. With Professors James R. Larus and David A. Wood, he co-founded the Wisconsin Wind Tunnel project that has developed new methods and new designs for parallel computer systems. After earning tenure, Hill went on sabbatical to Sun Microsystems where he worked on high-end servers. Hill is now Professor and Romnes Fellow at the University of Wisconsin-Madison, Information Director of ACM SIGARCH, and a Senior Member of the IEEE.

References

[1] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared-Memory System, to appear in *Proceedings of the IEEE*, special issue on distributed shared-memory systems, 1999.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer, special issue on shared-memory multiprocessing*, pages 66–76, December 1996.

[3] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

[4] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992. Parts of this work originally appeared as IBM technical reports in 1984 and 1985.

[5] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 13th Ann. Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[6] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

[8] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report #61, SCI Committee, March 1989. Also available as Computer Sciences Technical Report #1006, University of Wisconsin, Madison, February 1991.

[9] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. on Computers*, pages 175–189, February 1983.

[10] Mark D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, to appear in 1998.

[11] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. 19th Ann. Intl. Symp. on Computer Architecture*, pages 13–21, 1992.

[12] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[14] Robert H. B. Netzer and Barton P. Miller. Detecting Data Races in Parallel Program Executions. *Research Monographs in Parallel and Distributed Computing, MIT Press, 1991.*, August 1990.

[15] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 12–23, 1996.

[16] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, April 1988.

Retrospective:

Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors

Kourosh Gharachorloo

Western Research Laboratory
Digital Equipment Corporation
kourosh@pa.dec.com

The memory consistency model influences many aspects of shared-memory multiprocessor system design, including the design of programming languages, compilers, and the underlying architecture and hardware. The choice of the model can have significant impact on *performance*, *programming ease*, and software *portability* for a given system. Models that impose fewer constraints offer the potential for higher performance. At the same time, fewer ordering guarantees can compromise programmability and portability.

Our initial paper on memory models was written over eight years ago, at a time when cache-coherent shared-memory multiprocessors were not widely believed to be scalable. The Stanford DASH project was one of the pioneering efforts in this area and provided an excellent infrastructure for studying fundamental issues related to scalable shared-memory systems. Our research on memory models led to a number of other papers that addressed issues such as performance, efficient implementations, and better abstractions and formalisms for capturing the semantics of various models. We refer interested readers to the thesis for a comprehensive coverage of these topics [5]. A short, easy-to-read tutorial paper is also available [2].

It is always interesting to look back at a paper after many years to see which ideas or observations have withstood the test of time. We begin by describing what we consider to be the key contributions of our initial paper. We next present an up-to-date assessment of the key issues related to memory consistency models.

Key Contributions of Paper

Our initial paper contributes a number of key ideas. First, the distinction between read (acquire) and write (release) memory operations used for synchronization captures the basic intuition about the ordering imposed by such synchronization: an acquire relates to memory operations that follow it and a release relates to memory operations that precede it in program order. In addition to the reordering of operations between an acquire and a release, this observation enables the reordering of previous operations with an acquire and future operations with a release. Today, this form of reordering is exploited by virtually every program that runs on commercial shared-memory systems with relaxed models (even stricter models allow write-read reordering, whether or not either operation is used for synchronization).

The second contribution is a formal abstraction, in the form of properly-labeled programs, that captures the types of optimizations exploited by relaxed models within a simple and easy-to-use programming style. To enable safe optimizations, the programmer conveys high-level information (e.g., through labels or higher level primitives) about the behavior of memory operations, such as whether an operation is involved in a race with other operations to the same address. This approach is referred to as *programmer-centric* to contrast it with the *system-centric* way that directly exposes the programmer to the low-level reordering optimizations. Any program can be properly labeled without requiring structural changes (such as additional synchronization) by simply providing correct labels. Allowing conservative labels greatly simplifies the task of labeling and permits the programmer to focus on providing accurate information in performance critical regions. Another important benefit of the programmer-supplied information is automatic and efficient portability across a wide range of implementations [5].

The remaining part of the retrospective describes relevant developments since the publication of our initial paper. The discussion includes lessons that we learned the hard way (e.g., complexity of dealing with specifications and proofs in this area), findings that surprised us (e.g., relative performance of models on some platforms), developments we didn't expect (e.g., clever implementations for stricter models, or the large impact on software DSM research), and developments we expected that haven't materialized yet (e.g., relevant compiler and programming language research). We also cover the trends in commercial systems.

Current Memory Models

We briefly mention some existing relaxed models, that we will refer to throughout, broadly categorized based on how they relax program order constraints relative to sequential consistency (SC). The first category of models, which includes Sun's total store ordering model (TSO) and processor consistency (PC), allow a write followed by a read to execute out of program order. The second category, which includes Sun's partial store ordering (PSO) model, allows writes to be reordered also. Finally, models in the third category additionally allow reads to execute out of program order with respect to following reads and writes; examples are weak ordering (WO), release consistency (RCsc/RCpc), Digital's Alpha model (Alpha), Sun's relaxed memory order (RMO), and IBM's PowerPC model (PowerPC). More information about these models can be found in [5].

Formalism and Correctness

It is far too easy to underestimate the prevalence of subtle yet important correctness and performance issues that arise in specifying and implementing memory consistency models. Therefore, it is absolutely critical to use a formal and precise specification framework.

While the "perform with respect to" framework by Dubois et al. (referenced in our initial paper) was commonly used and seemed sufficiently formal at the time, we later realized that it has several shortcomings. For example, phrases such as "uniprocessor control and data dependence are obeyed" turn out to be ambiguous (Section 4.6 in [5]). Similarly, the framework is not general enough to capture some key optimizations, such as a processor reading its own write before the write is serialized with respect to other processors [9].

To remedy these issues, we developed a formal framework for specifying the ordering constraints imposed by a model [5]. Our specifications have the additional advantage of inherently exposing aggressive optimizations by imposing as few constraints as possible. We have specified existing memory models in this uniform framework, and have used the specifications to determine automatic and efficient mechanisms for porting programs across different models.

Programmer-Centric Approach

We extended our properly-labeled framework (partly in collaboration with Sarita Adve and Mark Hill) with an additional level of information about synchronization operations that enables safe use of optimizations captured by models such as PC, TSO, PSO, and RCpc. Proving these equivalences (e.g., SC=PC=RCpc) was more difficult than we originally thought, and benefited from the proof techniques developed by Adve [1].

This work led to a hierarchy of programmer-centric models that successively exploit more information about memory operations to achieve higher performance [5]. We also provide a set of sufficient conditions for supporting these models, which expose some additional optimizations beyond those allowed by release consistency. Overall, the most important information supplied by the programmer remains to be whether an operation is competing (i.e., involved in a race) or not. The extra levels of information are only important in programs with frequent competing operations.

Performance

Our performance studies of architectures with blocking reads led to observations that were somewhat surprising at the time, emphasizing the need for quantitative analysis of real workloads. We found that the ability to reorder reads with respect to previous writes is typically sufficient for eliminating the write latency in most applications [6], allowing models such as TSO and PC to perform as well as more relaxed models such as RC. This effect arises because (i) write misses tend to be finely interleaved with read misses (unlike the example in Figure 5 of the initial paper), and (ii) there are typically more read misses than write misses, providing the write buffer with ample time to retire the writes without filling up.

We also studied the performance of architectures with non-blocking reads in the context of dynamically scheduled processors (which are commonplace today). We found that the ability to reorder reads with respect to future operations (as in WO, RC, Alpha, RMO, and PowerPC) allows for hiding a substantial fraction of the read latency as well [8]

Implementation

The most interesting development in this area has been the emergence of techniques that exploit the fact that correctness is maintained as long as it *appears as if* the ordering rules for a model are obeyed. We proposed two such techniques, based on hardware prefetching and speculative reads, that can be easily added to dynamically scheduled processors to boost the performance of stricter models [7]. Even with sequential consistency, these techniques make it possible to exploit many of the reordering optimizations that were once believed to be allowed only by the most relaxed models. A number of commercial microprocessors (e.g., HP PA-8000, Intel Pentium Pro, MIPS R10000) have already adopted these techniques, and simulation results have confirmed that stricter memory models can greatly benefit from such techniques [11]. (Interestingly, one of my co-authors in [7, 8] believed that dynamic scheduling techniques were unlikely to be used in commercial microprocessors!)

Software DSM Research

Much of the recent research in software distributed shared memory systems has been dedicated to relaxing memory consistency models further and developing protocols that aggressively exploit such models (e.g., [4]), especially since such optimizations are important for alleviating false sharing in page-based systems. Furthermore, most systems have opted for somewhat restrictive synchronization models based on a limited set of system-defined primitives. This has led to a divergence with aggressive hardware relaxed models, making it difficult to port programs written for hardware multiprocessors to these software platforms. More recently, alternative approaches have been developed that allow transparent and efficient execution of hardware binaries [12].

Compilers and Programming Languages

This remains to be the least explored area with respect to memory models. Virtually all interesting compiler optimizations require the flexibility to reorder memory operations, and models that allow only some reorderings (such as TSO, PC, PSO) are not flexible enough (Section 5.10 in [5]). There has been some work on exploiting more reorderings under sequential consistency for a restricted class of programs [10]. However, research in compilers and programming languages has for the most part ignored correctness and programmability issues with respect to multiprocessor memory models.

Impact and Trends in Commercial Systems

Much has changed since the DASH project began. Scalable shared-memory machines are now available from several vendors. Aggressive implementation techniques such as lockup-free caches and dynamic scheduling (which we had to simulate for evaluation) are now commonplace. Most hardware vendors have opted for memory models that are more relaxed than sequential consistency. Machines using Sun and Intel processors primarily use TSO-like models, while others have opted for more aggressive models (e.g., Alpha and PowerPC). Relaxed models have proven themselves in the marketplace, with large bodies of software (including fully functional operating systems, compilers, and applications) developed for them. Awareness about the importance of memory models has also been elevated, with several vendors providing more precise descriptions of their memory models as part of the architecture specification.

A few companies have chosen to support sequential consistency (e.g., SGI). Nevertheless, all vendors (often implicitly) depend on relatively aggressive memory models to enable common compiler optimizations for explicitly parallel programs, hence exposing those programmers to relaxed semantics anyway. In addition, programmers who write sequential or implicitly parallel programs (i.e., automatically parallelized by compiler) are not exposed to the underlying hardware memory model and yet can benefit from implementations that support a relaxed model.

In contrast to aggressive models proposed in academia (WO and RC), commercial models (i.e., Alpha, PSO, RMO, PowerPC) convey ordering information through explicit fence instructions as opposed to using operation labels. Even though operation labels convey more information, this choice is understandable because of the difficulty of incorporating labels into existing instruction sets. In addition, most commercial models have opted to support atomic behavior for all writes (which leads to simpler semantics). This is a viable approach given the prevalence of invalidation-based cache coherence protocols and the ease of supporting atomic writes in such protocols. Commercial models also have to deal with non-trivial issues such as specifying the ordering semantics of I/O device operations, exception events, etc. (Section 4.5 in [5]). Finally, commercial models remain system-centric. This is primarily because programmer-centric models do not specify semantics for all possible programs and are considered to be specified at too high a level to convey a specific architecture. Nevertheless, the programmer-centric abstraction can still be used to simplify the task of reasoning about commercial models; for example, the high-level program information can be used to determine where explicit fence instructions are needed (Section 4.4 in [5]).

Commercial fence-based implementations can be improved in a number of ways. First, an explicit fence incurs an extra instruction fetch, and often has a non-negligible latency associated with it. It is important to minimize such costs especially for applications with frequent fences (e.g., database workloads [3]). Second, fence-based implementations do not provide an efficient platform for programmers who insist on stricter models such as sequential consistency. To remedy this, it is relatively easy to provide separate modes in an implementation (e.g., specified on a per process basis) that would implicitly impose the extra orderings required by the stricter model.

Advice for Aspiring Memory Model Designers

Coming up with a new memory consistency model is quite trivial. After all, it is just a matter of picking and choosing the set of orders that must be maintained among shared memory operations. The real design challenge lies in providing a *balanced* solution. Choosing or designing a relaxed model requires considering questions such as:

- What is the target environment, including the types of programmers, applications, and architectures?
- Are the semantics of the model defined precisely? How difficult is it to reason and program with the model? How restrictive is the model with respect to different programming styles?
- What are the practical implementation optimizations that motivate the model? How difficult is it to efficiently implement the model?
- What are the performance gains (at hardware and compiler level) from using this model relative to alternative models? And most importantly, do the performance gains justify the additional programming and implementation complexity?

Addressing the above issues is non-trivial since it involves considering complex trade-offs and interactions, in addition to dealing with subtle correctness issues.

Our hope is that the framework and intuition developed by our research pave the way for future advances in this area.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993. Available as Technical Report #1198.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66-76, December 1996. Extended version available as Western Research Laboratory Research Report 95/7.

[3] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[4] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144-155, May 1993.

[5] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1995. Also available as Western Research Laboratory Research Report 95/9.

[6] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245-257, April 1991.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In P*roceedings of the 1991 International Conference on Parallel Processing*, pages I:355-364, August 1991.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 22-33, May 1992.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Revision to "Memory consistency and event ordering in scalable shared-memory multiprocessors". Technical Report CSL-TR-93-568, Stanford University, April 1993.

[10] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *Conference on Program Language Design and Implementation*, pages 196-204, June 1995.

[11] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12-23, October 1996.

[12] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

RETROSPECTIVE:

Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

Western Research Laboratory
Digital Equipment Corporation, Palo Alto, CA 94301
jouppi@pa.dec.com

Motivation

The work in this paper was initially motivated by the BIPS project at Digital Equipment Corporation's Western Research Lab. This project started in 1988 with the goal of building a processor that could execute over one billion instructions per second [1]. As an early part of the design process, the performance of various possible system configurations were simulated. We found that for many configurations most of the potential performance was being lost in the memory system. This led to an effort to create a more efficient memory system given the expected resources (i.e., transistor counts, pin bandwidth, etc.) that would be available.

Just prior to this work, Mark Hill had been investigating tradeoffs involving cache set-associativity at the University of California at Berkeley. Mark classified cache misses into the now famous three C's: conflict, capacity, and compulsory misses [2]. Mark also showed that direct-mapped caches could give better overall system performance than set-associative caches in some situations because they had a faster access time.

Many advances in science are the result of new methods of measuring things. In this case, Mark Hill's measuring of the causes of cache misses led to my investigations of ways to reduce each type of cache miss. The miss caches and victim caches presented in the paper were proposed to reduce conflict misses, assuming a direct-mapped cache was used for its smaller access time. Stream buffers were proposed as a method for reducing primarily compulsory and capacity misses. The net result was a memory system with higher performance for a given transistor count.

Elaboration

Immediately after the 1990 ISCA paper I wrote a follow-up paper and submitted it to ASPLOS-III. This paper showed four things. First, it presented an enhancement to stream buffers which could supply prefetch data from any position in the stream buffer, resulting better utilization of data in the stream buffer and higher stream buffer hit rates. Second, it demonstrated that stream buffers were more effective than any of the hardware prefetching techniques that had been discussed in Smith's cache survey [7]. Third, it showed that stream buffers were more effective in reducing cache misses than victim caches. Fourth, and perhaps most interesting, it presented the effective cache size increase resulting from adding multiple stream buffers to a baseline cache design. In many situations, the combination of baseline cache and stream buffers could give miss rates equivalent to those of caches many times larger than the baseline design. Since stream buffers can eliminate compulsory misses, for some larger caches there was no cache size that had miss rates as low as the baseline cache plus stream buffers. Just as adding stream buffers can make a cache appear larger, adding victim caches can effectively provide fractional amounts of cache associativity (e.g., a miss rate equal to a 1.2-way set associative cache). This paper was not accepted to the conference and I never revised it or sent it anywhere else. As part of writing this retrospective I've turned it into WRL tech note TN-53 and put it on the web (see http://www.research.digital.com/wrl/techreports/pubslist.html).

One point of confusion for the reviewers of the ASPLOS submission was something that has come up many times since. In the original ISCA paper, it was not directly and explicitly stated in the text of the paper that data values in the stream buffer or victim cache are not available for use by the processor in the same cycle that data accessed from the cache would be available. However, all the block diagrams in the paper show that the victim caches and stream buffers are only connected to the memory refill side of the caches, and the text states that a cache line can be transferred from the stream buffer or victim cache to the primary cache in the cycle following the cache miss. Since the appearance of the ISCA paper, many people have assumed that a large multiplexor exists in the cache access path of the processor, and that this large multiplexor can select between the result of the cache probe and the contents of a stream buffer or victim cache all within the cache access time. This is certainly not the case, as can be seen from the diagrams in the paper. If there was such a large multiplexor, the access of the primary cache (direct-mapped in these examples) would become much slower. By placing a multiplexor on the cache refill path it is moved out of the critical cache access path at the cost of another cycle of delay. This cycle of delay is much shorter than a full cache miss penalty, so misses served from a stream buffer or victim cache still result in a significant win.

Evolution

Stream buffers and miss or victim caches have appeared in a number of systems, although their use is by no means widespread. The most compelling application remains prefetching instructions with instruction stream buffers; this is because of the highly sequential nature of instruction miss streams. Instruction prefetch buffers have appeared in a number of microprocessor designs.

One data-side application was a combined 2KB miss cache and stream buffer which was placed on the HP PA7100 microprocessor [3] while the primary caches remained off-chip. In this system they wanted large off-chip caches for good performance on large application programs. Off-chip caches are difficult to make set-associative because of limited microprocessor pin counts, and there was not enough space on the die for large caches, so they placed a combined prefetch buffer and miss cache on-chip instead. Because the combined miss cache and prefetch buffer was on chip, it can be accessed in parallel with the off-chip cache.

The Cray T3D and T3E also used stream buffers on the data side, as a replacement for a secondary cache. Because many real numeric applications use non-unit strides, support for non-unit stride prediction and allocation filters to reduce the memory bandwidth requirements were studied by Palacharla and Kessler [4]. As a result of this study allocation filters were implemented in the Cray T3E.

More recently, stream buffers have been studied in the context of more modern processor designs. Farkas et. al [5] found that stream buffers were useful in statically scheduled processors with non-blocking loads and speculative execution. Although dynamically scheduled processors are better at tolerating unpredictable memory latency than statically scheduled processors, they too can benefit from stream buffers as Farkas et. al. showed in [6].

In [6] improved per-load stride prediction and a variation of stream buffers called incremental stream buffers were also proposed. Incremental stream buffers do not attempt to fill the whole stream buffer on a miss, but rather extend the number of lines they try to prefetch if earlier prefetches are used. This reduces the amount of bandwidth wasted in cases where prefetching far beyond the initial miss is not useful. However, it is much better for short streams than methods that require several subsequent misses before allocating a stream buffer. Moreover, for long streams the additional startup overhead is insignificant.

Futures

Although miss caches and victim caches are typically more popular ideas than stream buffers, stream buffers are a more important and lasting contribution. Even with the system configurations of the initial paper, stream buffers made a larger contribution to system performance. As caches get larger, the percentage of misses which are due to conflicts goes down while the percentage due to compulsory misses go up. This further increases the importance of stream buffers at the expense of miss and victim caches. With technology scaling, the latency ratio between off-chip and on-chip cache access increases. This makes sacrificing some cache hit speed for increased cache hit rates (i.e., implementing true cache set-associativity) even more worthwhile. Finally, more recent system innovations such as dynamic scheduling which reduce temporal and spatial locality of reference streams require set-associative caches for good performance.

As technology integration increases, I believe stream buffers will still be important for obtaining the best system performance for ever-limited cache resources. And I believe there is still room for further advances in hardware prefetching.

Acknowledgments

Keith Farkas provided helpful comments on a draft of this retrospective as well as insightful work on stream buffer enhancements.

References

[1] Norman P. Jouppi, et. al., "A 300MHz 115W 32b Bipolar ECL Microprocessor," in the *IEEE Journal of Solid-State Circuits*, November 1993.

[2] Mark D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Thesis, University of California Berkeley, 1987.

[3] Ehsan Rashid, et. al., "A CMOS RISC CPU with On-Chip Parallel Cache", in the *Proceedings of the 1994 International Solid-State Circuits Conference*, pages 210-211.

[4] Subbarao Palacharla and Richard Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", in the *Proceedings of the 21st International Symposium on Computer Architecture*, pages 24-33, April 1994.

[5] Keith Farkas, Norman P. Jouppi, and Paul Chow, "How Useful are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?" in the *Proceedings of the 1st Conference on High-Performance Computer Architecture*, January, 1995.

[6] Keith Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic, "Memory-System Design Considerations for Dynamically-Scheduled Processors" in the *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[7] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, 1982

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.

Retrospective:

IMPACT: An Architectural Framework for Multiple-Instruction Issue

Wen-mei W. Hwu

Computer & Systems Research Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801
hwu@crhc.uiuc.edu

Background

The IMPACT Architecture Framework project started in 1987 at the University of Illinois, Urbana-Champaign. The goal was to develop an architectural framework and its enabling compiler technology for instruction-level parallel processing microprocessors. The project was founded based on three fundamental assumptions about the future of microprocessor architectures that were rather controversial at the time. First, the level of intrinsic instruction-level parallelism (ILP) in general applications should be much higher than that accepted by most researchers in the 1980's. The accepted standard was that fewer than two instructions per cycle could be sustained in non-numerical applications. Second, the compiler's role would be critical at achieving the increased levels of ILP in general applications. At this time, most researchers viewed advanced hardware mechanisms, such as out-of-order execution and branch prediction, as the only means to extract increased ILP. These assumptions have both been shown to be true through extensive evidence generated by many academic researchers and industrial groups, including the IMPACT team. Finally, industry would be able to implement high issue rate microprocessors at very high clock frequencies. As of 1995, the third assumption has been verified by successful products such as the Digital Alpha 21164.

Fundamental Approach

The IMPACT project took a very different approach than that followed by industry at the time. Traditionally, computer architects would propose instruction architecture features and evaluate the merits with machine language programming. After the processor is designed, a compiler team would attempt to produce compilers that generate code of similar quality to that of the hand code. Unfortunately, this goal was seldom reached due to the inherently different natures of automatic compilation and hand machine code programming.

John Cocke at IBM and John Hennessy at Stanford pioneered co-design of compilers and instruction architectures in the early 1980's. Their work resulted in the first generation of RISC microprocessors where features such as sizable register files and versatile but primitive instructions allowed a backend optimizer to eliminate redundant computation and a register allocator to remove unnecessary memory accesses. The RISC work at IBM and Stanford was focused on making efficient use of a pipeline designed to execute one instruction per cycle.

The IMPACT project applied the same compiler and architecture co-design methodology to the design of multiple-instruction-issue microprocessors. During the development of the compiler technology, the IMPACT project team members identified several major inhibitors to producing code with a high level of ILP. Many of the inhibitors could be removed by clever compiler transformations. One of the inhibitors, however, turned out to be difficult to do without architectural support: exception conditions generated by control speculative instructions. The architecture support for this aspect became the core concept of the paper.

Ground Work

The amount of work done for the paper was tremendous. The first major hurdle was to establish the fact that the compiler base was as good as, or better than, existing product compilers. Typically, research compilers have two fundamental problems. One is the inability to compile large programs. Two is the lack of advanced optimization capabilities to generate code of comparable or better quality than production compilers. Over ten people-years went into establishing a base compiler. In the paper, the IMPACT compiler was able to compile UNIX utilities and generate MIPS code that ran faster on a DEC-3100 workstation than the code generated by the MIPS and GNU compilers at their highest level of optimization. In the process, the team ran into tedious issues such as compatibility with the MIPS assembler, which was not well documented for external users.

The second hurdle was to develop and verify compiler optimizations that utilize architecture features not available on existing machines. The new compiler optimizations focused on a suite of ILP enhancing optimizations to generate efficient code for multiple-instruction-issue processors. In addition, an instruction scheduler that aggressively employed control speculation to move instructions across basic block boundaries to achieve a compact schedule was constructed. The verification process used was to emulate the proposed architecture features by translating the code generated for the new hypothetical architecture into code sequences executable on the DEC-3100 workstation. The execution of the code sequences then generates a trace of the original hypothetical machine code to drive a trace driven simulation. Although the emulation was conceptually simple, it was extremely difficult to debug when handling large applications. Unlike most architecture experiments where traces were generated on vendor machines using vendor production quality software, the entire software and hardware emulation tool chain was produced by the IMPACT team. Any bug exposed by a benchmark in the tool chain makes it impossible to conduct the experiments with that benchmark.

This paper signified an important milestone in the IMPACT project. It was the first time the complete compiler in conjunction with the emulation and simulation tools had been successfully utilized to conduct a system level architectural experiment. The abilities to vary the machine model and retarget the compiler to utilize different architectural features was a novel experimental framework. The infrastructure that was developed for this paper served as the foundation for much of the future work done in the project.

Intellectual Contributions

It has been seven years since the paper was first written. From the numerous feedback received to date, two contributions consistently stand out. First the paper was the first to establish that a compiler can generate highly efficient code from general applications with enough instruction level parallelism to utilize a four-issue machine. It changed many people's minds about the viability of high issue rate microprocessors as general purpose machines. This is especially an accomplishment considering the fact that the mainstream microprocessors of the day the paper was written issued up to one instruction per cycle and the fact that many researchers published pessimistic studies on the available instruction-level parallelism using existing product compilers.

The second frequently mentioned contribution is the understanding that exception handling support for control speculative instructions is key to the success of instruction-level parallel architectures [1]. The non-trapping instructions were later added to several RISC architectures, including SPARC and HP PA-RISC. In the research arena, this paper was followed by a series of work on Sentinel scheduling that allow the hardware to accurately detect and recover from exceptions generated by control speculative instructions [2]. A similar feature was incorporated into the IA-64 architecture.

Potential Improvements

In retrospect, the paper could be improved with additional real code examples to motivate, illustrate, and analyze the proposed non-trapping instructions. In terms of experiments, the additional cache misses and TLB misses, and page faults due to speculative instructions should be characterized to provide more insight into the proposed features.

By-products

The IMPACT Compiler and IMPACT Architecture Framework Emulation Tools have become a heavily used research infrastructure at several major corporations. The IMPACT software has since been completely redeveloped to provide comprehensive support for predicated execution. Licenses have been issued to major corporations including Intel, Hewlett-Packard, Advanced Micro Devices, IBM, SUN Microsystems, and Lucent Technologies. They have been used by industry research and advanced development groups for the purpose of designing new microprocessor architectures. A new program at the University of Illinois was started in 1997 to release parts of the IMPACT software environment to academic research institutions worldwide.

Authors

Pohua Chang graduated in 1991 and joined the Intel corporation. He was the author of many original IMPACT compiler modules. He published numerous journal articles and conference articles in the area of architecture support and compiler techniques for control speculation.

Scott Mahlke further developed the IMPACT architecture and compiler support for predication and control speculation in his Ph.D. dissertation work. He graduated in 1996 and joined Hewlett-Packard Laboratories. He published numerous papers in the area of architecture and compilation support for predicated execution. He also has major publications on the subject of accurate exception detection and recovery from exceptions in control speculated code.

William Chen pursued data speculation in his Ph.D. dissertation work. He graduated in 1993 and joined the Intel Corporation. He has major publications and holds key patents in the area of data dependence speculation.

Nancy Warter pursued isomorphic control flow transformations in her Ph.D. dissertation work. She graduated in 1993 and joined the faculty of California State University at Los Angeles. She has published major conference and journal papers on the subject of supporting cyclic and acyclic code scheduling with isomorphic control flow techniques.

Wen-mei Hwu was promoted to the rank of Professor at the University of Illinois in 1996. He became an IEEE Fellow in 1997 for contributions to high performance compilers and microarchitecture. He also received the 1993 Eta Kappa Nu Outstanding Young Electrical Engineer Award, the 1994 Xerox Award for Faculty Research, the 1994 University Scholar Award of the University of Illinois, and the 1997 Eta Kappa Nu Holmes MacDonald Outstanding Teaching Award for his work in research and teaching.

Acknowledgment

The research reported in this paper was supported by the National Science Foundation under the Research Initiation Award program, NCR (Dr. Lee Hoevel), and AMD (Dr. Mike Johnson). Their generous support was critical to the quality of this work.

References

[1] "Three Architectural Models for Compiler-Controlled Speculative Execution", P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, *IEEE Transactions on Computers*, Vol. 44, No. 4, April 1995, pp. 481-494.

[2] "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution", S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, *ACM Transactions on Computer Systems*, Vol. 11, No. 4, November 1993.

RETROSPECTIVE:

The DASH Prototype: Implementation and Performance

Daniel E. Lenoski

Silicon Graphics
lenoski@sgi.com

James P. Laudon

ZSP Corporation
laudon@zsp.com

Our paper entitled "The DASH Prototype: Implementation and Performance" was given at the 19th ISCA in Gold Coast, Australia in May of 1992. This paper outlined our implementation experience and initial performance details of DASH, the first hardware implementation of the ccNUMA architecture. DASH was a large multifaceted research project at Stanford University led by John Hennessy, Anoop Gupta, Monica Lam, and Mark Horowitz. The overall goal of DASH was to break the scalability barrier of bus-based SMP machines and provide the massive parallelism of distributed memory while maintaining the shared-memory paradigm. While there had been previous switch-based SMPs built in the early 1980s (e.g., the Cray X-MP, Univ. of Illinois Cedar, BBN TC-1000, and IBM RP3), DASH added hardware support for global cache coherence. Hardware cache coherence improved processor performance and removed the burden of coherence from the user or compiler. During the late 1980's our group was not alone, there were efforts at MIT (Alewife and J-Machine), University of Wisconsin (Multicube), Encore Computer (Gigamax), Kendall Square Research, and the IEEE Scalable Coherent Interface (SCI) standards effort, but ours was the first to build a hardware implementation of this new class of machine.

The high-level structure of DASH was a collection of nodes, each including one or more processors and a portion of the global memory, connected by a scalable interconnect (a 2-D mesh). Directory-based coherence, originally proposed by Censier and Feautrier in the late 1970s, was employed since it removed the need for the global bus found in snoopy systems. While the original directory schemes used a central memory/directory, moving to a distributed organization scaled memory bandwidth naturally with the number of processors. With this fundamental system structure in mind, and previous studies showing the potential of distributed-directories (see the Agarwal/Hennessy paper on directories in this collection), work began on the DASH prototype.

DASH Prototype Goals and Timeline

Scaling the cache-coherent SMP model to hundreds of processors raised many questions in the area of processor and system architecture, operating systems, compilers, programming languages, and parallel applications. We chose to build an actual hardware prototype of the architecture to address these questions as well as to:

- Understand the hardware complexities of actually building this type of machine.
- Provide more insight into the performance attributes of a real ccNUMA machine.
- Allow a comparison of real applications' complexities and performance (not simply small simulated kernels) among highly parallel shared-memory programs and their message-passing counterparts.

The difficulty of implementing a distributed directory protocol was of serious concern since it amounts to replacing the software controlled network interfaces on message-passing machines with hardware control for sending network messages to fetch remote memory and maintain cache coherence. At the time, it wasn't clear if this hardware complexity was tractable, and even if it was, would the performance of a ccNUMA be competitive with message-passing systems?

We began detailed architecture work on the system in fall of 1988. Early on, we made a choice to leverage an existing SMP system as our base node because these machines provide the necessary hooks for controlling the processor caches from their bus interfaces. We were anxious to utilize a RISC-based SMP, and the recently announced SGI 4D/240 series was the only such machine on the market at the time. This choice turned out to be very fortuitous, since utilizing an existing system allowed us to leverage much of the system hardware and software and concentrate our efforts on the unique ccNUMA hardware and software.

Initial power-on of the prototype system was in the Fall of 1990 and a 16 processor system was stable in the Spring of 1991. We then started work on a larger 64 processor system, which resulted in a stable 48 processor prototype in the Spring of 1992. Nagging problems with our ribbon-cabled mesh links prevented us from reaching the goal of 64 processors in a single system (a 4x4 mesh of 4 processor nodes), but we were able to learn much from the 48 processor prototype.

Innovations in DASH

During the architecture phase of the project, our focus was on the coherence protocol and mechanisms that would minimize memory latency and maximize memory bandwidth. In addition, we realized that hiding memory latency would also be key since the distributed structure of a large ccNUMA would invariably lead to longer memory latency. Likewise, support for large-scale parallelism demanded that we pay attention to synchronization and inter-processor communication. Being one of the first to tackle these problems in the context of a ccNUMA machine, these goals led to many innovative solutions. These included:

- Software-controlled non-binding cache line prefetch to hide latency and increase memory pipelining.
- Release-consistency support with fence/memory barriers to help hide store latency.
- Queue-based test-and-set locks to allow efficient contended spin-locks.
- Fetch&Inc and Fetch&Dec (borrowed from the NYU Ultracomputer, but without combining) for support of efficient barrier synchronization and distributed queues.
- Update coherence and deliver instructions which provide low latency inter-processor word and cache line communication respectively.

The actual hardware implementation phase also demanded innovative solutions such as:

- An efficient “forwarding” coherence protocol which minimized latency for accessing dirty data and writing to shared cache lines.
- Support for both invalidate and update coherence within the same directory protocol.
- Separate request and reply paths that prevented dead-lock on the normal memory requests together with retry mechanisms that handled race conditions in the distributed directory protocol.
- A high-bandwidth DRAM directory access path which performed read-modify-write cycles under the shadow of the main memory’s fetch of 16-byte memory blocks.
- One of the first lock-up-free caches that implemented a remote access cache to track outstanding memory references and supplement the processor caches with features such as prefetch.

Lessons Learned

As one would expect, building and using the DASH prototype led to many new insights and lessons that were both positive and negative. The most positive result was that it was feasible to build a ccNUMA machine and to achieve good performance on highly parallel shared-memory applications. Furthermore, by analyzing the logic in the directory and network interface, the prototype demonstrated that adding hardware cache coherence added only 10% additional hardware over a non-coherent MPP system structure. Another lesson was that with close attention, it was possible to keep remote-to-local memory latency to within a 3 to 1 ratio. Several features included in the prototype proved very successful. Operations such as prefetch proved to be very powerful in hiding memory latency and improving the pipelining of memory operations. Fetch&Op performed at memory also greatly reduced the overhead of barrier-type synchronization by reducing the serialization time for atomic counter operations.

Other features that did not yield as much performance improvements as expected were queue-based locks and update and deliver operations. While these operations could greatly aid in specific low-level communication, the overhead of general communication associated with inter-processor data sharing tended to swamp out the incremental enhancements that these operations provided. This was especially true on the prototype hardware where our remote access cache was as close as we could get the data to the processor (thus reducing latency by no more than a factor of 3).

Another somewhat unexpected result was the negative impact of using a bus-connected inter-node interface. Since memory operations needed to cross the processor's local bus twice and memory home's bus once, the resulting memory bandwidth when all processors are accessing remote memory was no better than one-third that of local memory. In fact, DASH's bus-bandwidth was a greater limit on global remote memory bandwidth than network bisection bandwidth. While not inherent to ccNUMA systems, this issue illustrated the limitations of simply extending a bus-based SMP with a ccNUMA network interface card.

The advantages of leveraging an existing SMP was also one of the indirect, but very positive, lessons from the DASH project. Using an existing SMP allowed a small university team to focus their attention on the important task of architecting and designing the hardware necessary to implement a ccNUMA machine. In addition, the choice of the SGI 4D/240 as the base node helped in the quick development of DASH, as its modest level of integration (by today's standards) allowed us to work primarily at the board level with PALs and FPGAs. Working at this level of integration reduced both design and debug time. There were some compromises due to leveraging an existing machine, but it reduced the time from concept to running real applications under Unix to less than 2.5 years. This increased the impact of the actual DASH hardware and helped validate the feasibility of the ccNUMA architecture.

Conclusions

The impact of the DASH project has been felt both in the academia and industry. Scalable shared-memory multiprocessors continue to be a hot topic of research. DASH helped validate the viability of the ccNUMA approach and provide a baseline to evaluate improvements in coherence protocols, scalable directory storage, and alternative system architectures such as COMA. ccNUMA systems have now been commercialized by a number of vendors including HP/Convex, Silicon Graphics, Sequent, HaL, and Data General. The DASH prototype helped pave the way for these commercial developments by detailing many of the fundamental design problems with ccNUMA machines and demonstrating that the shared-memory paradigm could be scaled and realize both good performance and good cost-performance.

Building the DASH prototype would never have been possible without the hard work of a number of individuals. These included our co-authors: Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. Additional contributions to the hardware development were made by Kourosh Gharachorloo, Wolf-Dietrich Weber, Mark Horowitz, Tom Chanak, John Maneatis and Monica Lam. Help from Silicon Graphics, namely Jim Barton, Forest Baskett, John Burger, Doug Solomon and John Carlson, was also instrumental. Dan Lenoski was supported by Tandem Computers during his graduate work. John Toole and Gil Weigand at DARPA provided the funding to support the greater team and build the DASH prototype.

For additional details on DASH see:

[1] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance.," *IEEE Trans. on Parallel and Distributed Systems*, 4(1)41-61, January 1993.

[2] D. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, San Francisco, CA 1995

Retrospective:

Active Messages: A Mechanism for Integrating Computation and Communication

Thorsten von Eicken
Cornell University
tve@cs.cornell.edu

David E. Culler
University of California at Berkeley
culler@cs.berkeley.edu

Klaus Erik Schauser
University of California at Santa Barbara
schauser@cs.ucsb.edu

Seth Copen Goldstein
Carnegie Mellon University
sethg@cs.cmu.edu

The impact of Active Messages on the many facets of parallel computing surprised even us. Surely this success had an element of articulating the right concept at the right time. Like most good ideas, in hindsight it was present in some form or another in many places, and it certainly had roots in the architectural discussion of the time. Simplicity, flexibility, high-performance, combined with a clear cost model, were key in the acceptance of the Active Messages approach in many arenas. The underlying idea was simple: each message names a handler at the destination and on message arrival the handler is executed with the message as an argument. The handlers must execute quickly and cannot block. Their sole purpose is to extract the message out of the network and incorporate it into the on-going computation. Active Messages provides flexibility since handlers can be customized to particular communication instances, or less generally, to communication protocols or programming models. Active Messages implementations achieve high-performance because they eliminate complicated buffer management and simplify deadlock considerations.

Active Messages grew out of our work of compiling implicitly parallel dataflow languages, in particular Id90, to commercial large-scale parallel machines. Such languages require extremely fine-grained communication and therefore a communication infrastructure supporting very low overhead messages is essential. In the 1990-91 time frame, industry was building interesting machines on the scale of a thousand processors, but the architecture community took surprisingly little notice. Mostly these machines were used to demonstrate that message passing costs were very high, but would be much lower on proposed designs — just as soon as they got built. We did not want to wait. Moreover, in looking closely at the commercial architectures, the hardware was clearly capable of far better communication performance than what was delivered. Thus, Active Messages was born of expediency: we sought to define a simple, flexible communication primitive that would match what large parallel machines actually did well. This would serve as an instruction set extension for communication and be used as a compilation target for languages that allowed fine-grain communication.

We had a straightforward goal: we wanted to measure interesting programs on a large scale and were ready to compile all the way down to the network on large parallel machines to cut down the communication overhead. The challenge we faced was that the basic communication operations available needed to deal with a large number issues, including routing, address translation, protection, buffering, output buffer full, deadlock freedom, event notification and more. Those thousand instructions in the commercial message passing layers were doing quite a bit. Although many machines could initiate a transfer to or from the network in a couple instructions, it took extremely careful engineering to address all of these issues in only a few more. While working on Active Messages, the handler-based messages of the J-machine and Monsoon were fresh in our minds, but our view was that the communication instruction set should operate with fixed storage resources.

There are several reasons for the success Active Messages enjoyed. Timing and technological advance were certainly important factors. Having completed the nCUBE implementation, we were in a unique position to appreciate and exploit the capabilities of the CM-5 as soon as it appeared. Getting the overhead down to 2 μs made Active Messages on a commercial machine competitive with the contemporary research machines, and it was only a matter of downloading a small library. Within a few months of release it was installed on CM-5s throughout the world. Later, it was incorporated into commercial products by Thinking Machines, it was implemented on the Paragon and incorporated into the OSF release, and recently it was incorporated into the IBM SP product. AM became the basis for much of the high performance cluster work and the recent standardization effort in the Virtual Interface Architecture. Without the caliber of the initial implementation and the willingness to distribute and support the code, Active Messages would probably not have enjoyed the impact it has.

The universal communication mechanism underlying Active Messages provided a flexible tool for building the substantial protocols involved in message passing and shared memory programming models. Several research groups needed just such a tool. Extensions of CC-NUMA designs were being considered with elaborate protocols, or even application specific protocols. Active Messages gave them a common vocabulary and a framework for tackling this problem. A protocol could be defined by a collection of handlers and triggered by a hardware initiated request message. The debate was on what the handlers did and whether a subset of them was cast in silicon. New parallel object oriented languages were being developed around C++; Active Messages gave them an efficient remote method invocation. Many new libraries were being developed to provide distributed data structures, numerical routines, scheduling, and so on; here too Active Messages provide a qualitatively better implementation technique than traditional message passing. Even the MPI effort to standardize conventional message passing had to explain that it was not exposing Active Messages to its users.

We did not expect Active Messages itself to become a programming model; it was intended as a compilation target and as a means of implementing the protocols associated with programming models. However, many programmers preferred to use Active Messages directly. The basic remote procedure call implemented by Active Messages was convenient. It also made it easy to reason about the inherent communication costs of a parallel algorithm because there was little happening outside the programmer's control. Such programmatic use, rather than the controlled usage within a communication library or compiler run-time, created a tension for the evolution of Active Messages. If the storage model was relaxed or handlers ran as first class threads they could do much more, however, the same level of performance and predictability could not be assured. Also, whether handler execution was truly interrupt driven or just implicit in touching the network came to be an issue, because the ways of dealing with variables that are shared between handlers and computation was slightly different in the two models. General application use lead us to provide a generic Active Message interface, rather than a specific one for each platform, as when it was viewed as a communication instruction set.

One thing that stands out in our minds is that much of the success of Active Messages came from our research model: the actual building of systems influenced by both compiler writers and architects. Although the ideas in the paper for how AM would be implemented on a processor have not come to pass, it has influenced parallel languages, compilers, machines, libraries, networks, network interfaces, and theory. The Threaded Abstract Machine that drove our view of compiling to the network continues to influence work on multithreading and light-weight threads packages and most message passing libraries have an AM-like underlying transport layer. Probably the most clear legacy will be the Virtual Interface Architecture (VIA) that promises to bring low overhead communication to clusters of PCs and workstations. VIA follows UNet in forgoing the remote handler discipline in favor of exposing the remote queues, but retains the view that user applications have direct, protected access to the network and can build up their own protocols. In the coming years we will see the impact of fast communication on large commercial applications.

The "implement to understand" point is worth reiterating. Active Messages was not born as a "neat idea" that was then implemented to evaluate its performance. We only understood which features really were as simple as we thought when we tried to implement a number of alternatives. Also, we were only compelled to develop Active Messages after carefully analyzing the implementations of alternatives by other researchers. The complexities of the dataflow model, of the message driven processor, or of distributed cache coherence state machines only become apparent in real implementations. In the near future we will get a wealth of experience with mainstream applications on top of VIA and memory-based interfaces to fast cluster interconnects, while the transistor budget continues to explode, compilers technology focuses on dynamic languages with broad use of threads, and network routers become more active. It seems clear that the question of how to efficiently integrate communication with computation will continue to be a crucial one.

We would like to thank the agencies and individuals that made such a difference in the impact of Active Messages. The work was supported by a collection of "exploratory" funding sources: the NSF PYI program, an IBM fellowship, Sandia Laboratory, and a Semiconductor Research Corporation fellowship. We managed to squeeze time on the nCUBE at Sandia; nCUBE loaned us a little machine and helped out with the Vertex kernel; Thinking Machines Corp. found a way to get an early machine to UCB at a discount and Moose answered our questions at midnight PST. The flame was kept burning by several people that built follow-on Active Message layers, including Alan Mainwaring, Adam Greenberg, Rich Martin, Lok Tin Liu, Chad Yoshikawa, Chris Scheinman, Anindya Basu, and Matt Welsh. We thank our good friends at MIT for the work on Id90, Monsoon and the J-Machine, which stimulated this investigation.

Retrospective:

The Turn Model for Adaptive Routing

Lionel Ni

Department of Computer Science
Michigan State University, East Lansing, MI 48824
ni@cps.msu.edu

When the Caltech Cosmic Cube [1] was implemented in 1981, it triggered a new wave on parallel processing and refreshed interest in hypercube topology. The research community has studied various direct network architectures, especially the k-ary n-cube, and their topological properties. Based on the underlying graph-theoretical model, many new theories and properties, such as routing paths and graph embedding, were discovered for various network topologies.

From the practical aspect, the demand of low communication latency had inspired the design of new switching mechanisms. In 1985, the wormhole routing (now also called wormhole switching or cut-through switching) was implemented in the torus routing chip [2]. While the wormhole routing can significantly reduce the communication latency, it can also introduce a unique deadlock situation, which is quite different from those in other traditional switching mechanisms, such as the store-and-forward switching. Although the concept of virtual channels was proposed in [3] as a possible approach to avoid deadlock, early multicomputers used fixed routing paths, mainly based on dimension order routing, to avoid deadlock due to cost and performance reasons from virtual channels or multiple physical channels. The channel dependence graph model [3] was considered as the theoretical foundation to develop deadlock-free routing algorithms.

In January 1991, I offered a graduate-level course on Advanced Computer Systems at Michigan State University. Due to our past research interest in multicast communication on multicomputers, message routing was a focus in the course. When studying the 2D mesh network, I mentioned that if we could double the number of channels in both X and Y dimensions, it would support fully adaptive routing. As a homework problem, students were asked to prove that the adaptive routing in 2D mesh networks could be supported by doubling only the channels in either X or Y dimensions. Students had to find out a total ordering of those channels to prove the deadlock-free property. As an open question, students were asked to think about what is the minimum number of channels required to support adaptive routing.

Chris, now Dr. Glass, was a Ph.D. student looking for his dissertation research topic. Due to the similarity between the routing algorithm and his interest in the street-walking algorithm (how to walk most quickly from one location to another along the rectilinear streets of a city), Chris chose this topic as his term project. In March 1991, he completed the technical report entitled "Adaptive, Deadlock-Free, Wormhole Routing in k-ary n-cubes." The concept of turn model was first presented in this report. Chris attempted to use a systematic approach to develop the most flexible routing algorithms without adding extra channels. This elegant turn model then became the core of his Ph.D. dissertation research. He developed a flit-level network simulator to further verify and evaluate different routing algorithms. We were surprised to find out that under uniform traffic, adaptive routing does not provide better performance than static dimension order routing. We first thought that something was wrong with the simulator. However, we then realized that there was nothing wrong with the simulator or our results. In his dissertation research, Chris demonstrated how to apply the turn model to various topologies. In the case of hypercube, the turn model helped to develop a new partially adaptive routing algorithm. He also showed the influence of various input and output channel selection policies on the network performance. Finally, he proposed the first fault-tolerant wormhole routing algorithm that can handle dynamic faults [4].

Significant research has since been engaged in the design of new and improved adaptive wormhole routing algorithms, fault-tolerant wormhole routing algorithms, and routing algorithms for other network topologies including irregular network topologies. Some new theories were proposed. The most notable one was the sufficient and necessary condition for deadlock-free wormhole routing [5]. Some aggressive routing algorithms based on deadlock detection and recovery were proposed. The turn model was also applied to the development to multicast routing algorithms. This research area has grown rapidly and almost reached a level of maturity. A comprehensive treatment of these important topics and engineering issues in the design of interconnection networks can be found in the recent book [6]. Adaptive wormhole routing was adopted in later generation parallel computers. Research interests in this area have moved to other important topics, such as efficient network interface design, and to other related areas, such as cut-through switch architecture in high-speed networks.

References

[1] C.L. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-33, January 1985.

[2] W.J. Dally and C.L. Seitz, "The torus routing chip," *Journal of Distributed Computing*, vol. 1, no. 3, pp. 187-196, October 1986.

[3] W. J. Dally and C.L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. on Computers*, Vol. C-36, No. 5, pp. 547-553, May 1987.

[4] C.J. Glass and L. M. Ni, "Fault-tolerant wormhole routing in meshes," *Proc. of the 23rd International Symposium on Fault-Tolerant Computing*, pp. 240-249, June 1993.

[5] J. Duato, "A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks," *IEEE Trans. On Parallel and Distributed Systems*, vol. 6, no. 10, pp. 1055-1067, October 1995.

[6] J. Duato, S. Yalamanchili and L. M. Ni, *Interconnection Networks: An Engineering Approach*, IEEE Computer Society Press, 1997.

Retrospective:

Alternative Implementations of Two-Level Adaptive Training Branch Prediction

Tse-Yu Yeh

Intel Corporation,
Santa Clara, CA
tyyeh@mipos2.intel.com

Yale N. Patt

Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor, MI 48109
patt@eecs.umich.edu

The Two-Level Adaptive branch predictor was conceived at Michigan during October, 1990. At the time, we and Mike Butler, another Michigan Ph.D. student in the HPS research group, were collaborating extensively with Mike Shebanow, Mitch Alsup, and Hunter Scales, all of Motorola, on a paper showing that Instruction Level Parallelism was greater than two [1].

The collaboration was initiated by Mike Shebanow, a designer of Motorola's MC88120. Shebanow was one of the original inventors of the HPS execution model, which attempted to obtain performance by wide-issue instruction supply and multiple deep pipelines with out-of-order execution to prevent blocking. He had shown as early as 1984 that more than 1/3 of the potential performance of an HPS microengine was lost due to branch prediction misses, and had proposed [2] his Autocorrelation Predictor as a way to improve on the saturating two-bit up-down counter [3], which was the most accurate predictor at that time. As part of that collaboration, Tse-Yu Yeh and Mike Butler worked with Shebanow at Motorola the previous summer, and Yale Patt visited Motorola regularly. Our studies that summer, based on the HPS paradigm, confirmed that the amount of work that would be thrown away due to a branch misprediction was prohibitively far too large. Thus, anything less than a very aggressive dynamic branch predictor was unacceptable.

The outgrowth of that summer resulted in the Two-Level Adaptive Branch Predictor. It was first published in Micro-24, in November 1991 [4], followed by the more comprehensive study in ISCA-1992 [5].

Tse-Yu Yeh presented the concept at the University of Michigan Industrial Affiliates meeting (IPoCSE) in Ann Arbor, in April, 1991, with representatives of Intel in attendance. At the time, Intel was already strongly considering a wide-issue, deeply pipelined implementation of the x86 architecture, and knew that the two-bit saturating counter mechanism would not provide sufficient prediction accuracy. Their reaction to the Two-level predictor was one of excitement. They subsequently adapted the model to their needs in what came to be the Pentium Pro microprocessor. The Two-Level predictor has continued to evolve since its beginnings in 1990, by its originators at Michigan and by other researchers at many major university and industrial research centers. Pan et. al. [6] introduced the GAs predictor, which took advantage of correlation among branches in the same equivalence class. McFarling [7] modified the use of the history register for indexing into the Pattern History Tables, reducing negative interference. He called his branch predictor gshare. Nair [8] suggested the History Register keep track of the history of the path of previous branches, rather than the history of their directions. Chang [9] augmented the set of Pattern History Tables of two-bit counters with a table of target addresses to handle indirect branches. Several authors have suggested combining compile-time information with the dynamic predictor. Chang [10] suggested classifying branches at compile time so that the dynamic predictor would only be used on non-unidirectional branches, reducing interference. Sechrest [11] investigated the role of adaptivity in the PAg Two-Level predictor. Young [12] proposed using profiling and code restructuring to allow static prediction while achieving prediction accuracies approaching that of a dynamic Two-Level predictor. Recently, Evers [13] has begun to study exactly how many of the branches in the History Register really contribute to predictions, and which simply get in the way.

In summary, in 1990, it was clear to us that if the HPS paradigm, with its wide-issue instruction supply and multiple deep pipelines, was to be successful, then a very accurate branch predictor would have to be developed, since at that time, none existed. The result of our work was the Two-Level predictor. Today, the Two-Level predictor has been implemented in multiple commercial microprocessors, and branch prediction papers extending the predictor appear at virtually every major conference from research groups at many major universities.

Tse-Yu Yeh received his Ph.D. from Michigan in EECS in 1993 and has been at Intel since then. He is currently a microarchitecture manager working in the Merced project. Yale Patt continues to teach both freshmen and graduate students and direct the research of Ph.D. students at Michigan in high performance computer implementation. Mike Shebanow is now CTO and Vice President of HAL Computer Systems in Campbell, CA, where he is responsible for the development of very aggressive high performance microprocessors. The early research was supported by Motorola, NCR and Intel. Particular acknowledgment is due to Dave Mothersole of Motorola, Lee Hoevel, formerly of NCR, and Fred Pollack, Konrad Lai and Bob Colwell of Intel for believing in and supporting the early work.

References

[1] M.Butler, T-Y Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism is Greater than Two," P*roc. 18th International Symposium on Computer Architecture*, May, 1981.

[2] Michael C. Shebanow, "Autocorrelation Branch Prediction," unpublished technical report, 1984.

[3] James E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th International Symposium on Computer Architecture*, pp. 135-148, 1981.

[4] Tse-Yu Yeh and Yale N. Patt, "Two-Level Adaptive Branch Prediction," *Proc. 24th International Symposium on Microarchitecture*, pp. 51-61, 1991.

[5] Tse-Yu Yeh and Yale N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th International Symposium on Computer Architecture*, pp. 124-134, 1992.

[6] S.-T. Pan, K. So and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76-84, 1992.

[7] Scott McFarling, *Combining Branch Predictors*, Technical Report, TN-36, Digital Western Research Laboratory, June, 1993.

[8] Ravi Nair, "Dynamic Path-Based Branch Correlation," *Proc. 28th International Symposium on Microarchitecture*, pp. 15-23, 1995.

[9] Po-Yung Chang, Eric Hao and Yale N. Patt, "Predicting Indirect Jumps using a Target Cache," *Proc. 24th International Symposium on Computer Architecture*, pp.274-283, 1997.

[10] Po-Yung Chang , Eric Hao, Tse-Yu Yeh and Yale N. Patt, "Branch Classification: A New Mechanism for Improving Branch Predictor Performance," *Proc. 27th International Symposium on Microarchitecture*, pp.22-31, 1994.

[11] Stuart Sechrest, Chih-Chieh Lee and Trevor Mudge, "The Role of Adaptivity in Two-Level Adaptive Branch Prediction", *Proc. 28th International Symposium on Microarchitecture*, pp. 264-269, 1995.

[12] Cliff Young and Michael D. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.232-241, 1994.

[13] Marius Evers, Sanjay J. Patel, Robert S. Chappell and Yale N. Patt, "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work", *Proc. 25th International Symposium on Computer Architecture, 1998*.

RETROSPECTIVE:

The Cedar System

*A. Veidenbaum** *P.-C. Yew*† *D. J. Kuck***
*C. D. Polychronopoulos** *D. H. Padua** *K. Gallivan*††

* Authors are with CSRD of the University of Illinois at Urbana-Champaign (**Emeritus)
† Author is with the Dept. of Computer Science at the University of Minnesota, Minneapolis
†† Author is with the Dept. of Computer Science at Florida State University

Project Goals

The Cedar project was officially started in 1984 following many years of research by our group in parallelizing compilers, parallel algorithms, and vector and multiprocessor architecture. At the time, multiprocessing was not universally accepted as a way to speed up the execution of a single program. The primary goal of the Cedar project was to "demonstrate that the supercomputers of the future can exhibit the general-purpose behavior and be easy to use" [1]. We felt that major advances in the state of hardware technology, architecture, compilers, and parallel algorithms made such a demonstration possible. A two-phase approach was advocated: the construction of a 32-processor prototype followed by a production system with thousands of processors. We stated that "the prototype design must include the details of scaling the prototype up to a larger, faster production system". Both "architectural and technological upward" scalability was required. Another goal was to have the prototype "achieve Cray-1 speeds for programs written in high-level languages and automatically restructured" by a compiler. Finally, "an integral part of the design... was to allow multiprogramming".

1. The quotes are from the [GKLS83] reference in the 1993 ISCA-20 paper.

This research was supported by the Department of Energy under Grant No. DE-FG02-85ER25001 and by the National Science Foundation under Grants No. US NSF-MIP-8410110 and NSF-MIP-89-20891, an NSF PYI grant NSF-CCR-89-57310, and support from several industrial affiliates including Alliant, IBM, AT&T, DEC, and CDC.

General Programming Model

Cedar was to be a scalable shared memory multiprocessor to achieve programmability. To avoid problems with control of a parallel computation and high synchronization overhead that had proved fatal in some of the previous systems, "a hierarchy of control" was to be used and a macro-dataflow model of computation was defined. A program was to be decomposed into a set of tasks, a dataflow graph of the tasks built and directly executed. A ready task in the macro-dataflow graph could be scheduled on one or more clusters.

The architecture was to be constructed as a hierarchy as well and consist of "processor clusters", each with a local memories, switch, and "synchronization unit". The clusters were connected to shared memory and had an ability to overlap cluster computation with shared to local memory moves. The "processor cluster" was to be a basic schedulable unit. The memory was to be organized as a hierarchy and a compiler was to assist in managing it. In particular, a form of software-managed caching of shared memory data in cluster local memory was to be provided.

Original Architecture Requirements

The architecture definition of the Cedar system was driven by the above considerations. The system architecture requirements were distilled to the now familiar list below.

- *Shared memory.* To achieve high bandwidth it was to use:
    - an "interleaved" design, separate from cluster memory and with its own network,

  - high-bandwidth, low latency interconnect. A multistage design was to be used to avoid latency problems of the earlier, mesh-connected computers.
- Support for multiple levels of program parallelism
- *Efficient synchronization and scheduling support* via a processor in memory
- *Memory hierarchy,* with software-controlled data "caching" in cluster memory
- Data and code prefetching
- *Scalability* to a larger number for processors

Implementation

Some original implementation ideas were modified in the course of the project in light of schedule and/or hardware constraints. For example, hardware control of macro-dataflow scheduling was shifted to software with efficient hardware synchronization support. Some of the more complex memory-based synchronization primitives were not implemented. Finally, the degree of memory interleaving was reduced by a factor of 2. Even so the resulting board size, packaging complexity, and the need for new surface-mount packaging pushed the limits of technology and delayed project completion.

Architecture/Hardware

The emergence in 1983-84 of small, high performance multiprocessors, such as Alliant and Elxsi, allowed us to avoid building our own "clusters" but also limited some of our architectural options. The Cedar team designed and built the Omega networks, shared memory/synchronization processor boards, network interface boards, and performance monitoring hardware. Our hardware linked four Alliant "clusters" into the shared-memory multiprocessor. The resulting system led the introduction of many new architecture and software ideas now commonly found in scalable MPs and even small symmetric MP systems. These are described next, followed by our retrospective view on improving them.

- *Shared memory.* Word-interleaved, UMA shared memory was part of the virtual address space and directly accessible via processor instructions. It was not cached and was physically separate from cached "cluster" memory. Shared memory is now implemented in three out of four U.S.-made scalable MP systems, with message passing architectures in retreat. If allowed to change one thing about our implementation it would be to provide a direct path between shared and cluster memories, which was difficult due to our use of Alliant clusters.
- *Shared memory-based synchronization*. Read-modify-write indivisible operations commonly used in bus-based system are extremely inefficient in large-scale systems due to interconnect latency and contention. Cedar implemented Test-And-Set and other Fetch-and-Op primitives via a fast processor in memory. A 32-bit Fetch-And-Add operation cost two extra clocks over a regular read.
- *Omega network interconnect*. The network used a buffered, self-routing design. Measurements on the actual hardware helped to better understand the effect of multi-word requests, such as vectors or cache lines, and of a processor's limit on the number of outstanding requests on network performance. Simple traffic throttling at the network interface would have been the most valuable addition. This type of network is now used in IBM's SP2 systems.
- *Per-processor strided block prefetch unit and a tagged storage buffer.* This unit, operating independently of its processor could fetch and re-order up to 8KBytes of data. It was limited by lack of address translation. Prefetching is now widely supported by commercial scalable MPs. The most valuable change would have been to prefetch into cluster memory as originally planned and to use multiple prefetch units/buffers.
- *Weak shared memory consistency.* To improve performance, Cedar allowed buffering of reads, writes and prefetches and had out of order memory request completion due to the interconnect. The interface hardware "blocked" the issue of synchronization operations until all previously issued shared memory requests completed. Relaxed consistency models are now widely used in commercial systems.
- *Use of clusters*. Alliant was chosen for its basic processor performance and similarity in programming model, the latter due to Alliant's founders having been influenced by interaction with our group. An 8-processor Alliant "cluster" efficiently supported vector/parallel computations and we extended parallelism by another level. As a result, Cedar could support three levels of parallelism, memory, and synchronization hierarchy in hardware.

Software

The Cedar system software supported an explicit parallel API which included nested DOALLs, task management, data placement as private/shared, and synchronization. Many of the

API features have now been incorporated in OpenMP, a portable parallel programming standard. Standardization of the interface will undoubtedly help to speed up the acceptance of parallel programming.

A single system image for parallel programs was provided by the Xylem OS kernel running on top of a cluster's Unix system. Xylem implemented global, dynamic task creation and scheduling the assigning of tasks to available clusters. A single shared memory virtual address space was supported, across all clusters, while private memory in each cluster was individually managed. File systems in each cluster were made available to the user as a single abstraction. The single system image OS for parallel computers is still a research topic and is another area where standardization would help make parallel processing more usable.

Cedar automatic compilation goals were very ambitious and were only partially met during the project. Many of the ideas have since been implemented in other compilers for both uni- and multiprocessors. In particular, automatic detection of parallelism and management of the memory hierarchy made great strides during and since the project. The concept of compiler-based coherence was widely explored and program restructuring to optimize for a cache hierarchy is now widely used.

Macrodataflow ideas of the Cedar project led to many novel scheduling algorithms for arbitrarily nested loops and for general task-graph program models. A more lasting impact came from some of our research work on the efficient and transparent combination of parallel processing and multiprogramming (or equivalently, multithreading and multiprogramming). Results of that work were incorporated in a number of commercial compilers and operating systems, with the latest being the SGI Irix 6.5.

In addition, parallel algorithm design during the project made great strides in understanding and optimizing for a complex parallel/memory hierarchy, with some of the ideas now appearing in compilers and numerical libraries. For example, the BLAS3 were first implemented for the Cedar linear algebra applications.

Performance evaluation

At the start of the project, Livermore loops were a standard benchmark used by the community. An important effort spearheaded by the Cedar staff was the creation of a user group to define a standard set of application benchmarks. This led to the "Perfect Club" benchmark suite. The group was later merged with the SPEC consortium and the codes became the first version of the SPEC-HPG benchmarks. Cedar performance was carefully evaluated using these and other benchmark applications/algorithms. It led to better understanding of architecture, compilation, and application programming through hardware and software performance monitoring. The original performance target of the project, Cray-1 performance on a wide range of high-level language programs, was demonstrated. In many cases performance scalability matched that of the later Cray model, 8-processor YMP.

Finally, during the project a set of practical parallelism tests was defined to quantify how well a parallel system performed. In particular, they stressed delivered performance, scalability, and programmability. These tests are still very relevant today and having today's multiprocessors pass the tests remains a great challenge.

Conclusion

The multiprocessor design space is very large and has not yet been sufficiently explored. We believe there is a continuing need to study scalable MP systems, especially in light of current and future hardware and software capabilities. Architectural ideas cannot be studied in isolation and require an interdisciplinary team of hardware, architecture, system and application software experts to collaborate. Such a team cannot settle for simulation studies; it needs to develop and experiment with real systems. Testing ideas via implementation, especially their interdisciplinary verification is a key to developing successful architectures. In particular, this avoids the benchmarking syndrome of replacing all application/algorithm experience with simple codes. It also allows the actual concerns, modes of operation, and preferences of one group of experts to be communicated to the others concretely. Finally, 'overlapped' development and research nurture each other and greatly accelerate progress. Most of the above happened on the Cedar project and was both a major achievement and a key to our success.

The potential to speed up program execution by increasing the number of processors continues to hold great promise. This has now been demonstrated by several large production systems on a number of applications. It is also becoming common-place as a cost-effective means of enhancing desk-top performance via parallelism. However, widespread use of parallel programs is still being hampered by programming and performance tuning complexity, architectural bottlenecks, and by their high cost additionally limiting availability of scalable systems. Overcoming these obstacles remains a grand challenge.

Retrospective:

Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer

Matthias A. Blumrich *Kai Li* *Richard D. Alpert*
Cezary Dubnicki *Edward W. Felten* *Jonathan Sandberg*†

Department of Computer Science,
Princeton University, Princeton, NJ 08544

†Morgan Stanley,
1585 Broadway, New York, NY 10036

Introduction

This paper is the first paper reporting the SHRIMP project at Princeton [1]. It describes the design of a network interface that implements a virtual memory-mapped communication model for protected, user-level communication.

The vision of the SHRIMP project was to investigate how to use commodity PCs and commodity software (including operating systems) as building blocks to construct scalable, inexpensive servers that can deliver performance comparable to or better than custom-designed multicomputers. A challenge was to deliver communication performance to applications that was close to the hardware limit. This paper was the result of our effort to design network interfaces to support protected, user-level communication.

In this retrospective, we would like to describe where our ideas came from, how the research project started, how we wrote the paper, how well the ideas worked out, and the current state of the SHRIMP project.

Lessons from the PRAM Project

The most significant influence on our network interface design was the lessons learned from the Pipelined RAM (PRAM) project [2, 3], an effort from the Massive Memory Machine project at Princeton. The PRAM prototype used a physical memory-mapped network interface and a broadcast switch to connect multiple machines together.

The prototype PRAM network interface used 32 Kbytes of dual-ported SRAM, where one port of the SRAM connected to the I/O bus and the other connected to the broadcast network. A 4-port broadcast network switch was built, and a cluster of four PCs was operational between 1987 and 1990. The one-way latency to move data between the SRAM memories of two 386 PC's over the PRAM network interfaces and the broadcast switch with optical links was about 10 microseconds.

Kai Li and Jonathan Sandberg discussed lessons learned from the PRAM effort on several occasions in 1989 and 1990. They felt that the memory-mapped communication model is good because it is simple and imposes very little overhead. On the other hand, physical memory-mapped communication has several limitations. First, it does not allow multiple processes to use the network simultaneously. Second, the PRAM network interface used an <address, data> pair for every remote update. Therefore, the communication mechanism was inefficient for large block transfers since half of the bandwidth was used by addresses. Third, using a relatively small dual-ported SRAM on the network interface made the implementation simple, but required copying on both the sending and receiving sides to support applications. Fourth, PRAM offered a variant of shared memory, but it was not very easy to program. Finally, it is difficult to scale up a broadcast network.

These lessons led us to develop a list of requirements (or a wish list) for the network interface we would like to build. We thought an ideal network interface should have the following features. First, it should use a virtual memory-mapped communication model, instead of a physical memory-mapped communication. The virtual memory-mapped communication model can take advantage of the virtual memory management unit to support multiprogramming. Second, the network interface should transfer data between the host main memories (as opposed to dedicated memories on the network interfaces). Third, the network interface should combine updates to consecutive addresses into a single packet to utilize the network well. Fourth, the network interface should support an efficient, remote DMA mechanism that could move data from one virtual address space to another across the network. Fifth, one should use a modern routing network rather than a broadcast network. Finally, the network interface should support shared virtual memory well. This long list laid the foundation of the network interface we designed for the SHRIMP multicomputer.

Starting the SHRIMP Project

A combination of several factors started the project in the fall of 1992. At a DARPA PI meeting at Daytona Beach in September 1992, Chuck Seitz gave an inspired talk on the chipsets his research group was trying to build at Caltech, based on their Caltech Router and Mosaic multicomputer. Afterwards, David DeWitt, Kai Li, Richard Lipton, and Jeffrey Naughton talked about building PC clusters using a scalable network. They got together with then DARPA program managers Brian Boesh and Gil Weigand to discuss the idea, and received encouragement to submit a white paper. At the DARPA PI meeting, Kai Li spoke with graduate student Matthias Blumrich about the vision of the project. Blumrich was very interested, and decided that he wanted to design the network interface hardware. Blumrich and Li started investigating the tradeoffs involved in using Caltech routers, including chip testing and switch designs.

Kai Li presented the idea to several people at Intel including Paul Close, George Cox, Konrad Lai, Fred Pollack, and Justin Rattner, and discussed the tradeoffs of using the Caltech routing chips, the Intel Delta multicomputer routing backplane (built with the Caltech routers) and using the Paragon multicomputer routing backplane. The best choice was to use the Paragon routing backplane and Intel Supercomputer Systems Division agreed to provide the routing backplane and technical help.

Kai Li, Richard Lipton, David DeWitt and Jeffrey Naughton wrote a white paper and submitted it to DARPA at the end of 1992. This white paper contained the original vision of the SHRIMP project, design tradeoffs, and a project plan. A colleague of DeWitt and Naughton, Michael Carey, contributed the acronym SHRIMP for Scalable High-performance Really Inexpensive MultiProcessor. According to the white paper, the PIs at Princeton would lead the effort of building the multicomputer, and the PIs at Wisconsin would use it in their parallel database research.

We began the design work on the network interface at the end of 1992, though the project was formally underway in 1993. In that year, Cezary Dubnicki and Richard Alpert joined to help on the software side. Cezary eventually implemented most of the system software, and Richard implemented the NX message-passing library. In the fall of 1993, Doug Clark and Ed Felten joined Princeton faculty and started working together on the project.

We were developing two network interfaces at that time. The first one was called SHRIMP-I, a "quick-and dirty" design that would allowed us to rapidly build a system to support virtual memory-mapped communication [4]. We soon found out that it was too much of a distraction in a relatively small project to have two hardware efforts, so we discontinued the SHRIMP-I effort. The second network interface is the one described in the paper and has been sometimes referred to as SHRIMP-II, or just SHRIMP.

Writing the Paper

The network interface for the SHRIMP multicomputer contains everything we had in the requirement list mentioned above. Towards the end of 1993, the design was complete and we had detailed hardware simulation with Mentor CAD tools.

We had several discussions about whether we should write the paper at that time or wait until the hardware was working. The main argument against writing the paper was a desire to wait until we had some experimental experience. There were several arguments for writing the paper. First, we wanted to share our ideas in a timely fashion with the ISCA community, where several groups were creating related projects (UC Berkeley started their NOW project toward the end of 1993, for example). Second, the network interface design was complete and we had simulated the hardware with CAD tools. Third, we had also built a simulator based on PVM for software development and used it to show that the communication model worked well. Based on these arguments, we decided to write the paper.

Douglas Clark contributed to the network interface design but he decided to remove his name from the co-author list. He felt that his contributions were not substantial, though we felt differently. In retrospective, we should have insisted further having him as a co-author.

Did the ideas hold up?

There are several ideas in the paper. The main idea was to use the network interface to support the virtual memory-mapped communication model. This idea was successfully confirmed with working prototypes later [5,6].

The idea of using automatic update to perform "zero-overhead" communication and to support shared virtual memory also worked out well, according to our experience implementing the AURC shared virtual memory protocol [6,7].

After publishing the ISCA '94 paper, we found that although the deliberate update mechanism that used a single instruction to initiate a data transfer was efficient, it was too restrictive in some ways. Most significantly, it required a static, many-to-one mapping from send buffers to a receive buffer, whereas many libraries and applications need the ability to initiate a data transfer from any send buffer to any remote receive buffer, at any offset. We later changed the implementation with a technique called User-level DMA or UDMA [8], which requires a two-instruction sequence to initiate a protected user-level block data transfer, but remedies the limitations of the mechanism described in the paper.

The SHRIMP project served as an umbrella for a lot of varied systems research at Princeton, work that was described in several ISCA papers [5,6,9] as well as papers published elsewhere.

Now the cycle is turning again. Drawing on the lessons of SHRIMP, we have built new systems based on more modern PCs, SMPs and Myrinet network interface hardware, combined with a custom firmware implementation of the virtual memory mapped communication model [10,11].

Acknowledgments

The SHRIMP project is sponsored in part by DARPA under grant N00014-95-1-1144, by NSF under grant MIP-9420653, and by Intel Corporation.

We would like to thank Paul Close, George Cox and Justin Rattner for helping us access the routing network technology used in the Intel Paragon multicomputer, David Dunning and Roger Traylor for patiently helping us understand the details of the Intel MRCs and NICs, Konrad Lai and Wen-Hann Wang for helping us understand the memory subsystem of Pentium, and Malena Masarina for her help in designing the "SHRIMP-I" network interface. We also would like to thank David DeWitt, Richard Lipton, and Jeffrey Naughton for their help to start the research project, and Michael Carey for his contribution of the acronym SHRIMP.

References

[1] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten and Jonathan Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proc. of 21st Annual International Symposium on Computer Architecture*, April 1994. Pages 142-153.

[2] Richard J. Lipton and Jonathan S. Sandberg, "PRAM: A Scalable Shared Memory," Technical Report CS-TR-180-88, Princeton University, September 1988.

[3] Jonathan S. Sandberg, "The Design of the PRAM Network.," *The 2nd IEEE Symposium on Parallel and Distributed Processing*, December 1990. Pages 367-372.

[4] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li and Malena R. Mesarina, "Virtual Memory Mapped Network Interfaces," *IEEE MICRO*, 15(1): 21-28, February 1995.

[5] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damiankis, Cezary Dubnicki, Liviu Iftode and Kai Li, "Early Experience with Message Passing on the SHRIMP Multicompute," *Proc. 23rd International Symposium on Computer Architecture*, May 1996. Pages 296-307.

[6] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi and Robert A. Shillner., "DesignChoicesintheSHRIMPSystem:AnEmpirical Study.," *Proc. 25th Annual International Symposium on Computer Architecture*, June 1998.

[7] Liviu Iftode, Cezary Dubnicki, Edward W. Felten and Kai Li, "Improving Release-Consistent Shared Virtual Memory using Automatic Update.," Proc. 2*nd International Symposium on High-Performance Computer Architecture*, February 1996, pp. 14-25.

[8] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten and Kai Li, "Protected, User-level DMA for the SHRIMP Multicomputer," *Proc. 2nd International Symposium on High Performance Computer Architecture*, February 1996, pp. 154-165.

[9] http://www.cs.princeton.edu/shrimp

[10] Cezary Dubnicki, Angelos Bilas, Kai Li and Jim F. Philbin., "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet," *Proc. 11th Int. Parallel Processing Symposium*, April 1997.

[11] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefamos N. Damianakis and Kai L, "SHRIMP Project Update: Myrinet Communication," *IEEE Micro*, 18(1):50-52.

Retrospective:

The Stanford FLASH Multiprocessor

Jeffrey S. Kuskin

Computer Systems Laboratory
Stanford University
jsk@mojave.stanford.edu

Since the publication of the initial FLASH paper in the proceedings of ISCA 1994, the underlying architecture and goals of the FLASH system and of the MAGIC chip have remained essentially as described: to design a multiprocessor node controller (MAGIC) based on a flexible, programmable protocol processor core while minimizing the protocol processing overhead caused by using a flexible, rather than a hardwired, protocol engine.

Although this basic architectural goal remains, our perspective on the utility of flexibility has changed over the course of the design. In addition, as MAGIC and the other parts of the FLASH machine have moved from paper proposals to microarchitectural specifications to actual hardware — as of this writing a FLASH system is operational — we have developed some perspective on the trials and tribulations of building hardware in a university environment.

The sections that follow describe the current project status, our new insights on the benefits of flexibility, and offer some comments on how to best approach hardware construction in a university.

Project Status

Implementation of the MAGIC chip was the focus of the FLASH design effort. Initial plans called for a collaborative development effort with the Supercomputer Systems Division of Intel. The Stanford team was to handle the design, Verilog coding, and most of the verification, and the Intel team was to handle the physical implementation. Unfortunately, for business reasons rather than technical, joint development ended in early 1994 and the team at Stanford assumed responsibility for physical implementation of MAGIC. We selected LSI Logic's LCB500K ASIC process and, much later than hoped, sent MAGIC to LSI for fabrication in early 1997. The MAGIC die is 16x16mm, contains approximately 750K gates, and has 451 signal pins and 700 total pins.

We received MAGIC chips from LSI in late October of 1997 and successfully booted UNIX on the first silicon of MAGIC in January 1998. Multiprocessor UNIX booted soon thereafter. As of this writing, debugging of the multiprocessor configuration continues and plans to construct a 64-processor FLASH machine are underway. We have identified only a few minor bugs in MAGIC, most related to diagnostic functionality; none was serious enough to impede bring-up activity or compromise system operation.

Flexibility

The original paper argued that the main benefit of flexibility was that it allowed a single node controller design to provide integrated support for both message passing and cache-coherent shared memory. Integration of these two communication mechanisms offers the opportunity to combine in a single application the fine-grained, unstructured communication efficiency of cache-coherent shared memory with the bulk data transport and communication-computation overlap capabilities of a message passing system.

With a programmable protocol engine, MAGIC is able to support both of these communication mechanisms using the same underlying hardware simply by loading the appropriate protocol code. Indeed, integration of the two protocols is desirable even if programmers rarely employ both protocols in the same application. For example, the FLASH message passing protocol is built on top of the cache coherence protocol, allowing it to exploit the deadlock avoidance and performance optimizations present in the cache coherence protocol and avoiding the need to duplicate much of the lower-level protocol support code.

Although these observations about the benefits of flexibility are still valid, as the FLASH design progressed it became evident that flexibility offers a second, perhaps more important, benefit: it allows the node controller to adapt to the system size and application characteristics. Several performance investigations demonstrated that best performance on both small- and large-scale machines is not achieved with a single cache-coherence protocol and directory organization. Small-scale machines, for example, often benefit from a simple directory format — such as the ubiquitous bitvector — that minimizes protocol processing overhead. As the system size scales, however, the inherent quadratic growth in bitvector's directory memory requirements becomes untenable. Degradation into a coarsevector format sacrifices the ability to maintain precise sharing information and can increase message traffic and lead to a degradation in overall system performance. Rather than continuing to employ a bitvector-like directory format and protocol, then, larger systems are best served with a directory format — such as dynamic pointer allocation — that is inherently more scalable in terms of directory memory usage and can track sharing information precisely even at large processor counts.

In general, flexibility in protocol choice can allow a single architecture to achieve robust performance across a range of application and machine sizes. Because protocols differ substantially in their directory formats, inter-node messages, and implementation structures, providing a hardwired implementation of even a small set of protocols would consume an unacceptable number of gates. Instead, a flexible solution is desirable since it permits the system cache coherence protocols, directory formats, and other aspects of low-level inter-node communication to be tailored to the system size and, potentially, the application mix.

In addition to its ability to support a variety of communication protocols, MAGIC's flexibility also has proven extremely useful in system debugging. One of the first activities during FLASH system bringup was to implement an interface between a workstation and MAGIC via the FLASH node board's serial port. Protocol code executing on MAGIC's protocol processor can read from the serial port and write to it using a version of the C "printf" function that runs on MAGIC. This functionality has greatly aided bringup by allowing MAGIC to monitor its own operation and to detect and report errors to the bringup team.

The cache coherence protocol code, for instance, is augmented with normal C "assert" statements. A failing assertion invokes the printf code to report the problem, and the resulting error message appears in a window on the workstation. Indeed, much of the debugging capability present in the simulation environment used for MAGIC design verification is available in the actual hardware as well. Flexibility is a critical component of this feature, since it enables low-level testing and monitoring of MAGIC and of the protocols it is running without requiring dedicated hardware support and with the ability to remove any overheads associated with the additional debugging code when system operation is stable. These capabilities are likely to be extremely difficult to achieve in a hardwired node controller design without dedicating considerable hardware resources solely to debugging support.

Building Hardware in a University Setting

The design and implementation of FLASH was a large undertaking, particularly in a university environment. MAGIC's complexity and implementation technology are comparable to contemporary industrial projects; the SGI Origin 2000 node controller (the "Hub" chip), for example, was developed at the same time as MAGIC and has comparable gate count, ASIC implementation technology, and target cycle time.

Although completion of the MAGIC design undeniably took longer than we had planned, in the end the chip booted UNIX on first silicon, an achievement even by industrial standards. Several aspects of the MAGIC implementation effort have been especially helpful in allowing a small group of students — as is the case in most university hardware development efforts — to tackle such a large design. Surely other academic teams who have undertaken similarly ambitious (some might say foolhardy) designs could expand this list.

1. Focus on one design task. For us, this was the MAGIC chip.
2. For the rest of the design, leverage as much as possible from industry, adapting the design to accommodate what industry can provide. But make sure what is leveraged will be used and

actively supported by the industrial partner as well.

3. Ensure that at least some design team members have prior industry experience. Knowledge of general ASIC, printed circuit board, and system packaging design and manufacturing procedures is invaluable.

4. Take the time up front to build a solid, maintainable design infrastructure. This includes design automation tools, simulation and verification frameworks, design methodologies, and other tools that ultimately will improve the overall hardware design productivity and quality.

5. Take verification seriously. "If it wasn't tested, it won't work" is accurate. Resist the temptation to sacrifice verification thoroughness in the interest of finishing the design sooner.

6. Accept that the overall rate of progress in a academic setting will be slower than in industry. The comparative lack of resources, experience, and people inevitably leads to longer design times; schedule accordingly.

7. Ensure that the project has experienced, detailed, day-to-day management. If a faculty member plans to manage the project, be sure he or she appreciates the time commitment.

Despite these hurdles, designing real hardware is well within the capabilities of a university team. MAGIC was a long but ultimately very satisfying project, and the lessons learned from working under the constraints of actual hardware — particularly how these constraints affected the FLASH system architecture and the architecture and implementation of MAGIC — were invaluable.

Our ultimate goal is to use a larger FLASH system to significantly advance the capabilities of multiprocessor design research. FLASH permits a researcher to perform investigations on a real machine, using realistic problem sizes and an actual operating system, yet with the ability to control and monitor the lowest levels of system operation. These capabilities are not easily achieved with traditional simulation environments and are a key benefit of the MAGIC architecture.

RETROSPECTIVE:

Tempest and Typhoon: User-Level Shared Memory

*Steven K. Reinhardt**, *James R. Larus, and David A. Wood*

*EECS Department
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
stever@eecs.umich.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
wwt@cs.wisc.edu

Introduction

Tempest and Typhoon have emerged as among the most influential contributions of the Wisconsin Wind Tunnel project, a collaborative effort with Prof. Mark D. Hill, several staff members, and a large group of graduate students. This retrospective focuses on the origins of the Tempest and Typhoon ideas and their subsequent evolution.

The Beginnings

The seeds of the project began to germinate in late 1990 and early 1991 with our effort to rapidly prototype large-scale shared-memory multiprocessors. Because other research groups had a one- to two-year lead in their prototyping efforts—and considerably more resources—our project started with the goal of exploiting the parallel computers that our department was acquiring with funding from NSF's Institutional Infrastructure program.

During this exploratory phase, we made the essential observation that shared-memory systems permit a continuum of implementations, ranging from full hardware support to software simulation/emulation on a message-passing platform. Moreover, in the middle lies a rich collection of mixed hardware/software design alternatives.

An internal research note, dated July 9, 1991, roughly classified these alternatives into five levels:

Level 0: Software simulation/emulation. At this level, shared-memory programs execute on an unmodified message-passing parallel platform. A program's loads and stores are replaced with calls to routines that simulate the shared-memory behavior of the proposed design.

Level 1: Shared virtual memory. This level incorporates Kai Li's observation that address translation hardware can be used to map shared memory references to local pages and detect non-local references, albeit at coarse granularity.

Level 2: Fine-grain shared virtual memory. This level makes the observation that shared virtual memory can be implemented at a finer granularity given a mechanism—such as fine-grain "presence" bits—to detect when cache blocks are not stored locally.

Level 3: Local hardware support. This level begins to blur the distinction between a test-bed and a prototype. It extends level 2 with hardware support to initiate requests and handle responses on misses to remote data.

Level 4: Remote hardware support. The final level adds hardware support to handle external requests to a node's memory—that is, a directory controller. This last level encompasses all-hardware implementations.

Initially, we considered these approaches solely as alternatives for evaluating the hardware of interest, a highly integrated hardware-centric system. This discussion lead to the development of the Wisconsin Wind Tunnel (WWT), the parallel simulation system that gave our project its name [9]. The original version of WWT used a parallel message passing machine (a Thinking Machines CM-5) to simulate a hypothetical shared memory machine. WWT is a hybrid of levels 0 and 2, and uses the CM-5's ECC bits to implement fine-grain valid bits. Memory references that access non-local shared memory cause a trap, because of either a page fault or an intentionally set ECC error. Fine-grain access control allowed direct execution of shared-memory programs, which resulted in a very fast simulator that permitted rapid evaluation of hypothetical shared-memory implementations.

Cooperative Shared Memory

WWT was originally developed to evaluate an architectural approach called Cooperative Shared Memory (CSM) [4]. CSM's central premise was that hardware and software could cooperate to support shared memory efficiently. This cooperation took two forms. First, a programming performance model helped programmers identify expensive operations (so they could avoid them when possible) and helped hardware designers identify common cases (so they could optimize them). Second, CSM encouraged hardware designers to concentrate expensive hardware resources on optimizing frequent operations and to fall back to software for complex, less frequent cases.

Our programming performance model was called Check-In/Check-Out (CICO). It asked programmers to issue an advisory `check_out` directive before the expected first use of shared data followed by a `check_in` directive after the expected last use. We further proposed $\mathrm{Dir}_1\mathrm{SW}$, a minimal directory protocol that supported CICO-conforming programs efficiently (i.e., entirely in hardware). Violations of the CICO model, which often required more complex protocol operations, were handled correctly but less quickly by trapping to software. A later version of $\mathrm{Dir}_1\mathrm{SW}$, called $\mathrm{Dir}_1\mathrm{SW}$+, handled some common CICO violations in hardware as well [12].

Cooperative Shared Memory provided the philosophic underpinnings of Tempest and Typhoon. Hardware and software should cooperate to achieve good shared-memory performance. Programmers should be able to optimize performance by exploiting hardware mechanisms. Hardware designers should focus on providing efficient hardware mechanisms, and, as much as possible, leave policy to software.

WWT as a Shared Memory Machine

While designing and developing the Wisconsin Wind Tunnel, we met developers of the emerging generation of MPPs, the Intel Paragon and Thinking Machines CM-5. During these meetings, a frequent misconception was that WWT was a "real" shared-memory system, not just a test-bed. Students running programs on WWT also tended to blur this distinction.

In early 1993, we recognized that WWT was an interesting fine-grain shared-memory system in its own right, an observation that led to two parallel efforts. First, we began to develop a performance-oriented shared-memory system for the CM-5, simply by removing from WWT the components that calculated the performance of the hypothetical hardware. This effort led to the Blizzard systems (discussed further below).

Second, we realized that a small amount of hardware support might allow a message passing machine to achieve competitive shared memory performance. Our first step in this direction was a joint project with Thinking Machines and NimBus to develop an enhanced memory controller (EMC) that provided first-class fine-grain access control. The short-term goal was to eliminate the complex, relatively slow "hacks" required by WWT to manipulate ECC and synthesize a fine-grain read-only state via page protection. The longer term goal was to develop a "smart NI" that could handle the most frequent cases of a simple $\mathrm{Dir}_1\mathrm{SW}$-like coherence protocol—most likely with a programmable processor. The EMC chip was designed and fabricated by NimBus. Sadly, Thinking Machines never used it in a product, largely because of the additional product risk posed by the enhanced features.

Typhoon

Typhoon emerged as the follow-on to the EMC and "smart NI" approach. To minimize our exposure to Thinking Machines's marketing decisions, we envisioned a single ASIC that would not interfere with "normal" operations within a local node. The ASIC would provide hardware snooping support for fine-grain access control, an embedded protocol processor to implement some or all of the coherence protocol, and a closely coupled network interface.

A major goal of Typhoon was to increase programming flexibility beyond CSM, allowing programmers to optimize known communication patterns aggressively. The approach that we chose was to give programmers direct access to the raw mechanisms underlying shared-memory protocols. An important difference between Typhoon and our earlier $\mathrm{Dir}_1\mathrm{SW}$ work came from our realization that many protocols we envisioned needed flexibility on the requester side, not just on the directory side. This approach fit well with the "smart NI" model that called for using a programmable processor or controller to access the network interface. We refer the reader back to the original paper for the rest of the motivation and design.

Tempest

Programmers needed an abstraction of Typhoon's shared-memory mechanisms to develop protocols. Initially, we borrowed from the internal WWT interfaces and assigned each memory block an access control tag. Accesses that conflicted with the referenced block's tag invoked a user-specified handler. We initially referred to this abstraction simply as the "tagged block model".

Two important changes occurred in late 1993. First, we recognized the fundamental importance of the programming abstraction. The tagged block model applied equally well to the nascent all-software Blizzard system as to Typhoon, and it clearly made sense to support the same protocol programming interface on both systems. Although our original intent was merely to develop a simple abstraction for Typhoon, we ended up with a powerful abstraction for which Typhoon was just one implementation. Second, we gave the abstraction a "first class" name to reflect our appreciation for its importance. We chose Tempest, in keeping with the Wind Tunnel group's practice of naming nearly everything after a wind (fortunately, children have been unaffected by this practice).[1]

Subsequently, the Tempest interface [5] became the focus of much of the WWT group's research. Tempest's stable, powerful abstractions enabled parallel, synergistic research on both sides of the interface. On the system side, we began to explore the broad range of possible Tempest implementations. Other group members simultaneously investigated the implications of a flexible protocol interface for applications, programmers, and compilers. A key goal emerged to have Tempest provide application portability across a diverse range of implementations, each with different cost/performance objectives.

Blizzard: An All-software Tempest System

The Blizzard systems are a family of Tempest implementations that run on stock hardware [11]. One variant, Blizzard-E, uses WWT's "ECC hack" to provide fine-grain access control. Another variant, Blizzard-S, uses executable editing [7] to add explicit in-line checks. Both versions were initially implemented on the CM-5 and later ported to the Wisconsin COW, our cluster of 40 Myrinet-connected Sun SparcStation 20s. Our research on the applications of Tempest benefited greatly from the availability of a real (not simulated), relatively stable Tempest platform.

Typhoon-0: Minimal Hardware for Tempest

A key aspect of the Typhoon design is the (ab)use of existing snooping cache coherence protocols to provide hardware fine-grain access control on an otherwise unmodified platform. We decided to demonstrate the feasibility of this approach by implementing a prototype access control board for the Sun SparcStation and populating the 40 nodes of the COW. The resulting system, Typhoon-0 [10], can be viewed either as a prototype of the more highly integrated Typhoon or as a minimal-hardware Tempest implementation. Unlike Typhoon, Typhoon-0 relies on off-the-shelf devices for each node's network interface and protocol processor. In the process of our design and analysis of Typhoon-0, we recognized the benefits of an intermediate design, Typhoon-1 [10], that integrates the network interface (but not the protocol processor) with Typhoon-0's access control unit.

Custom Protocol Demonstrations

One of our early experiments investigated the performance gains made possible by writing custom, application-specific protocols [3]. The performance improvements for three application kernels on the 32-node Blizzard/CM-5 system ranged from 1.4–16 times, which strongly encouraged us to extend this approach. Subsequent experiments [8] also demonstrated the value of custom protocols in running parallel irregular applications. However, the efforts of many students showed that writing custom protocols was difficult and time consuming.

Programming Support

In response to these problems, the project investigated programming languages and tools to support custom protocol development. One effort lead to the Teapot language for writing and verifying custom protocols [2]. This language halved the size of a protocol, but more importantly, enabled use of automatic verification tools, drastically reducing the time and effort to produce a working protocol.

Another attack on the difficulty of writing protocols was to shift the burden of exploiting them from a programmer to a compiler. Several efforts clearly showed that compilers for high-level pro-

1. This practice created an unnecessary amount of confusion among the meteorologically challenged, who could not tell a Tempest from a Typhoon.

gramming languages could exploit custom protocols, to produce code with robust parallel performance that in many cases exceeded hand-written code. Initially, this work focused on research parallel languages, such as C**, in which Tempest supported a novel parallel programming model [6]. However, with the assistance of the Portland Group, we were also able to show that custom protocols could greatly expand the range of High Performance Fortran (HPF) programs that ran well [1].

Summary

The Tempest and Typhoon paper was the first of a broad collection of Tempest-related papers from the Wisconsin Wind Tunnel project (see `http://www.cs.wisc.edu/~wwt`). Its impact within Wisconsin has been considerable, contributing to 8 Ph.D. dissertations and 8 Masters degrees. We suspect its impact beyond Wisconsin has also been considerable, but we leave that evaluation to others.

Acknowledgments

Many have asked about the absence of Mark Hill as an author of the Typhoon papers. Mark has been a constant co-leader and contributor to the Wind Tunnel project, including the Tempest and Typhoon research. In mid-summer 1993, Mark unilaterally distanced himself from the Typhoon effort—against our objections—to help potential tenure letter writers differentiate our contributions from his own. Despite staying at arm's length, Mark made numerous contributions to this work and has been actively involved in the Tempest follow-on projects.

Many students have contributed to the Tempest work. We would like to single out one, Rob Pfile, for his efforts to implement the Typhoon-0 prototype.

This work received financial support from a number of sources. Initial support came from the National Science Foundations PYI/NYI program (grants CCR-9157366 and CCR-9357779). Primary support for Typhoon came from Michael Foster of the NSF's Experimental Systems program (grant MIP-9225097). Gil Weigand and Bob Lucas of the Defense Advanced Research Projects Agency supported the Blizzard implementations (ARPA Order Number B550). Dave Douglas, Bob Zak, and Greg Papadopolous provided technical and financial support from Thinking Machines Corporation and later Sun Microsystems. Additional support was provided by a Univ. of Wisconsin Graduate School Grant, a Wisconsin Alumni Research Foundation Fellowship, an AT&T Ph.D. Fellowship, and donations from Digital Equipment Corporation, Xerox Corporation, the Portland Group. Our Thinking Machines CM-5 and Wisconsin COW were purchased through NSF Institutional Infrastructure Grant CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

About the Authors:

Steven K. Reinhardt completed his PhD at the University of Wisconsin on the Typhoon implementations of the Tempest interface. He is currently an Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, where he is conducting research on parallel computer architectures and systems.

James R. Larus is an Associate Professor of Computer Sciences at the University of Wisconsin–Madison. His research includes programming languages and compilers, the design and programming of shared-memory parallel computers, program profiling and tracing, and program executable editing.

David A. Wood is an Associate Professor of Computer Sciences and Electrical and Computer Engineering at the University of Wisconsin–Madison. His research spans computer architecture, emphasizing parallel computer design, implementation, and evaluation.

References

[1] S. Chandra and J. R. Larus. Optimizing communication in HPF programs on fine-grain distributed shared memory. In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 100–111, June 1997.

[2] S. Chandra, B. Richards, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[3] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing'94*, pages 380–389, Nov. 1994.

[4] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993. Earlier version appeared in ASPLOS V.

[5] M. D. Hill, J. R. Larus, and D. A. Wood. Tempest: A substrate for portable parallel programs. In

Proceedings of COMPCON'95, pages 327–332, San Francisco, California, Mar. 1995.

[6] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, Oct. 1994.

[7] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[8] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[9] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[10] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[11] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–306, Oct. 1994.

[12] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in CMG Transactions, Spring 1994.

Retrospective:

The MIT Alewife Machine: Architecture and Performance

Anant Agarwal

Laboratory for Computer Science
Massachusetts Institute of Technology
agarwal@mit.edu

The MIT Alewife project evolved out of exploratory work at Stanford on directory schemes for cache coherence [1] (also included in this collection). Using data from small bus-based multiprocessors, this early work demonstrated that directory schemes were as efficient as bus-based snooping protocols, and that by distributing directories along with main memory, they could provide the foundations for a cache-coherent shared-memory multiprocessor based on an interconnection network. This paper further recognized the scaling limits of bit-vector directories — they consumed memory proportional to the square of the number of processors — and speculated that variants such as limited pointer directories or limited broadcast directories[1] might be attractive scalable alternatives. The paper, however, stopped short of demonstrating the feasibility of limited directories, largely because of the lack of either address traces or parallel programs written for a scalable coherent shared-memory system. This lack of data was not surprising given that such a machine had not been invented yet!

Exploration

The Alewife project was born out of a desire to build a *shared-memory* multiprocessor that was truly *scalable* (see the section "Perspectives and Summary" in the Alewife paper in the Proceedings of the Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, 1991, to get a sense of our early thinking). Although scalable message-passing multicomputers had been around for years, they were known to be notoriously hard to program. We believed that shared memory was easier to program, and accordingly, we chose early on to offer no compromise on the shared memory programming model.[2] Notice that our early Alewife thinking offered no plans to expose message passing to the software system.

For scalability, we chose to borrow heavily from the message passing machines conceived by researchers such as Seitz and Dally. Message passing machines achieved their scalability by distributing constant per-processor resources over a point-to-point interconnect and exposing this distribution to the programmer. Accordingly, we decided early on to distribute memory and processors over a point-to-point mesh network (as opposed to a uniform-access multistage network) and strove to keep per-node costs more or less constant. We believed that scaling to even tens of processors required support for locality management

1. A limited pointer directory maintains pointers to a fixed number of cached copies of data. A limited broadcast directory divides the processors into sets, and maintains a pointer to each set of processors, sending broadcast invalidations to the entire set when needed.

2. During the early Alewife days, the notion of shared memory with weaker memory semantics had not yet been formally defined. Therefore, we took sequentially consistent shared memory as a given. As discussed later, we chose to use context switching as a way of tolerating latency. When the weaker models began to appear in a sequence of path-breaking papers from USC, Wisconsin, and Stanford, we were faced with the choice of adopting a weaker model. At this point, we decided to support the sequentially consistent memory model since it did not require compromising the shared memory programming abstraction, and since our investigations revealed that the performance of weak consistency was comparable to other forms of latency tolerance.

from schedulers and compilers. As we discovered years later, (for example, see Nussbaum's PhD thesis), software management of interconnect locality in a cache-based system became important only for systems that exceeded many hundreds of processors.[3] We also learned later that the real benefit of mesh networks for few tens of processors was their low cost, modularity, and ease of packaging.

When the project started, we believed limited directories offered the solution to scalable memory requirements, since their per-processor costs scaled as the log of the number of processors. The discussion below tells why our initial optimism with regard to limited directories was completely misplaced, and how Alewife avoids their deficiencies.

Adhering to our shared-memory programming discipline, we wanted to avoid exposing locality to the programmer at all costs. Alewife does expose locality to the software system, and we believed we could develop limited-directory-based hardware and an accompanying software system that could exploit the underlying locality for scalable performance, while providing an uncompromising shared memory abstraction to the programmer. Our challenge was to build such a system and to demonstrate that the twin goals of programmability and scalability could be met.

Notice that in the very early Alewife days (Spring, 1988), the following features were chosen: a point-to-point mesh interconnect that exposed locality to the software, a limited directory distributed with memory among the processing nodes, and a shared-memory programming model. Features such as context switching, fine-grain synchronization, message passing, and in fact, the name Alewife itself, came later — each with an interesting history of its own. Our early research addressed two major questions, both related to scalability. Could shared memory systems tolerate the latencies of mesh interconnects? Could limited directories scale?

Although architects today do not think twice about using networks with non-uniform communication latency for shared memory, interestingly, we spent a lot of time worrying about their "programmability," that is, whether their non-uniform latency and bandwidth were mismatched with the demands of the uniform-access shared memory abstraction. Anecdotally, we were able to find some email from Agarwal to Hennessy in April of 1988 that talks of the tradeoffs in using a mesh network: "There are certainly a lot of problems [with mesh networks], and perhaps the main one is the issue of programmability. But if we are to get anywhere building large machines, this issue of locality (proximity) must be made visible to the compiler/scheduler (or to the programmer as in message passing machines or connection machines) in some graceful way."

Even more interestingly, the multithreading solution to the latency problem adopted by Alewife was inspired by the following response from Hennessy two days later. "When thinking about how to scale a shared memory machine above a few hundred processors, the difficulty becomes tolerating the latency. I tried to think how the message passing folks deal with their horrible latencies and it occurred to me — they context switch. Suppose you could build a machine that could context switch quickly (easy for a MIPS-style RISC machine, just use multiple register sets). Suppose you knew when a memory request would take a long time — simply context switch."

We chose to use context switching on cache misses as a mechanism to tolerate the latencies of mesh networks. We also began talking to Bert Halstead at MIT, whose group was exploring the design of a multithreaded processor called March. Like HEP, March used fine-grain multithreading (context switching on each cycle) to tolerate memory latency. March also included tag support for Multilisp Futures and fine-grain synchronization. Alewife's processor, Sparcle, (initially named April, for it followed March!), inherited many of the features of March, and improved upon it in many ways.[4] We believed single thread performance was key to the competitiveness of multithreaded processors. Accordingly, Sparcle context switched only on cache misses to remote memory and synchronization failures (both large latency events). Infrequent context switches allowed the architecture to exploit traditional pipeline optimizations for good single thread performance. It also enabled a simple implementation of Sparcle, since infrequent context switches are tolerant to relatively long context switch times (about 10 cycles).

3. A key reason is that even with efficiently engineered interfaces, a significant component of the remote access latency is attributable to overheads at the source and destination.

4. Kranz and Nussbaum worked on the March project and later joined Alewife.

Taking a minimalist approach, we also simplified the tag support architecture for fine-grain synchronization.

Realizing that building a multiprocessor system was a massive effort, we began to explore potential collaborations that could reduce our own effort. The first of such collaborations was with LSI Logic. We realized in the Summer of 1989 that the Sun Microsystems' SPARC architecture could yield a simple path to implementing the Sparcle processor (Sparcle was actually named following our decision to use SPARCs). We met with Gene Hill of LSI Logic, then the head of the SPARC division, and he agreed to help us implement Sparcle by modifying LSI's SPARC implementation. A fruitful collaboration with LSI and Sun followed this initial discussion.

We explored a collaboration with Tom Knight on interconnects. Knight was interested in developing a high-speed circuit-switched multistage interconnect, including a packaging technology using "fuzz button" pressure connectors in a liquid cooled system. Given our focus on communication locality, Knight offered to provide short-circuit feedback paths in the multistage interconnect to support fast near-neighbor communication in the multistage network.[5] We ultimately decided to stick with the mesh interconnect for many reasons. We felt the interconnect technology introduced too many additional failure modes into what was already an ambitious and risky project. The mesh network was a better match to Alewife's pedagogy. Early simulation results indicated that packet switching provided better performance than circuit switching for our system parameters. Finally, and perhaps most importantly, we were also able to obtain working Mesh Routing Chips (MRCs) from Chuck Seitz at Caltech, thereby eliminating (or so we thought) a major risk factor.

It turned out that the self-timed protocol of the MRC was both a blessing and a problem. It helped us in that we did not have to worry about clock synchronization across the entire machine. It also allowed us to conduct sensitivity experiments on the Alewife machine by varying the processor clock for the same network speed. These sensitivity experiments were critical in determining the ratios of processor to network clock speed under which either shared memory or message passing was optimal. Their asynchronous nature resulted in some nightmarish testing and debugging problems, turning many Alewife researchers into transmission line hackers. The asynchrony also required some creative test methodologies. Overall, we believe we came out well ahead by using the MRCs, and we are beholden to Chuck Seitz for making them available to us. On the other hand, the Alewife implementors will be very wary of asynchronous logic in the future.

By the Fall of 1989, the Alewife architecture had evolved to the following: Its fast context-switching Sparcle processors would be based on SPARCs and its mesh network would use Caltech MRCs. The Sparcle processor would support fine-grain full/empty bit synchronization. We, however, were beginning to weaken on the limited directory, and message passing had not shown up yet.

Design

Extensive simulations against address traces for large numbers of processors obtained by running several parallel programs from IBM, MIT, and Stanford during 1989 and 1990 began to lead us to the conclusion that limited directories were simply not robust. Although all programs exhibited predominately limited data sharing, disquietingly, almost every program included at least a small number of widely-shared (but mostly read-only) variables. Initially, we hypothesized compiler and software system passes that would automatically detect such widely shared objects and fix the problem, and thereby obtained fairly positive results out of our simulators by subtracting the effects of these errant references. For example, we believed we could eliminate widely shared variables in barriers by using scalable software combining trees.

Simulator hacks can only take you so far when you have undertaken to build a real working system, so we began to develop the hypothesized software passes needed for widely shared references. Unfortunately, each new program encountered a new type of optimization that had to be performed. The growing list of optimizations made the system extremely fragile, and gradually, our resolve weakened as evidence mounted on the fallibilities of limited directories. As a result of discussions with David James and Guri Sohi, we began to explore alternatives such as pointer

5. The name Alewife itself came up in 1988 during a discussion with Knight. Alewife was a recently constructed station on the red line in Boston's subway system. Knight's interconnect project continued under the name Transit.

chains that could still yield a constant cost per node. (Pointer chaining techniques, which were adopted for the IEEE Scalable Coherent Interface, linked cached copies using pointer chains rooted at the home memory node). We were concerned about the long latencies of single pointer chains and the complexities of the doubly linked alternatives. We also looked at purely software-based approaches using traps and software allocation of pointers in garbage-collected heap storage, and discarded them as being too expensive, at least for the processor-memory speed ratios at that time.

During this time of uncertainty in the project, Kubiatowicz had begun to design a system-level message interface so Alewife could perform I/O operations efficiently. It then occurred to us that we could take advantage of software-injected messages and a trap-based processor interface to extend the limited-directory mechanism into software in the rare case that a widely shared item caused a limited directory overflow. By gracefully extending the directory into garbage-collected heap storage and maintaining it as a software hash table with linked lists, we could allow widely shared objects to revert to the software structures. We named this scheme LimitLESS — limited directories locally extended with software support — and designed a unified message abstraction across both the software and the hardware. The LimitLESS scheme was particularly appealing because it enabled building an experimental system that allowed us to vary the number of hardware pointers from five to zero, zero being the all-software case in which all remote memory operations were being handled in software.

We believed that experimenting with the zero pointer case was important because it afforded a system with minimal hardware support for shared memory. As demonstrated by Chaiken in his PhD thesis, the all-software case was only about a factor of two or three off from the hardware case. Kirk Johnson's PhD thesis took the all-software approach one step further and explored the feasibility of a coherent shared memory system called CRL built on top of an efficient message passing substrate. Interestingly, this thesis articulates a key benefit of interrupt-driven delivery of messages: interrupts are better than polling when asynchronous messages invoke handlers that are unrelated to the computations being performed on the receiving processors.

Although the software-based LimitLESS approach had been conceived to solve the scalability problem of directories, we learned soon enough that it had other appealing properties such as flexibility and adaptability. Chaiken's thesis discusses several such adaptive protocols — for example, those that switch between individual invalidates and software broadcasts — and presents experimental data on their performance. Our instinct about the all-software case proved to be abundantly true as the flexibility and scalability of software approaches have all but shut out hardware directory approaches in more recent research projects.

The LimitLESS case study highlights perhaps the two most important reasons for building real systems in research environments. First, unlike simulators, real working systems seldom hide serious flaws. And second, when a research group has undertaken to build a novel system, they will invent the necessary mechanism and do whatever it takes to make it work. Such an environment of necessity that breeds invention is impossible to simulate.

The messaging interface changed the face of Alewife and rapidly established the value of integrating both messaging and shared memory. Recall, during the early Alewife days, shared memory and cheap messages were provided largely in exclusion in previous systems. Even in Alewife, messages were first introduced as a means of performing efficient I/O. They were then extended to provide the foundations for a software-based directory architecture. The message interface also allowed cost-effective solutions to deadlock problems caused by limited buffering in the network hardware. As the message mechanism was exposed to system software it rapidly pervaded the run-time system, since many operations such as scheduling and synchronization were best performed with messages.

Since the initial message interface was available only at system level, user-level software incurred a heavy overhead in using messages. The software folks campaigned for the same functionality to be available at user level, and convinced the architects to provide a user-level message send. Although it may seem that implementing both takes a kitchen-sink approach to the architecture, it turns out that shared memory systems require much of the underlying hardware functionality anyway. The additional requirements are to expose this functionality to the software. Kubiatowicz's thesis has a solid analysis of the extent to which resources can be shared between shared memory and message passing.

Implementation and Evaluation

We decided to use an application-specific integrated circuit (ASIC) for the Alewife cache and memory management unit (CMMU). This chip provided most of the hardware support for messaging and coherence. At this point we discovered that ASIC vendors were hardly tripping over each other to obtain our ASIC business. In fact, most ASIC vendors will not support a research chip project even with a high NRE (non-recurring expense — a one-time charge paid to the vendor) because the production chip volumes are usually quite low compared to the numbers they are used to. Continuing our relationship with LSI, we established contact with Brian Halla, then the general manager of the ASIC division at LSI (currently CEO of National), who graciously agreed to support our chip building efforts.

The chip design involved writing more simulators. The early phases of Alewife involved trace-driven simulators. These were replaced by ASIM, an instruction-level simulator. As the CMMU chip design started, ASIM was itself replaced by NWO (which stood for New World Order), which was faithful to the real design. NWO, in fact, incorporated some of the control logic directly from the real design. Testing and validation of the CMMU was done using LSI's simulators, augmented with a TCP interface to NWO. NWO also ran on Thinking Machine's CM-5, a configuration that facilitated software validations and architectural studies.

Looking back, the implementation effort was a process of incremental discovery in itself, and we happened upon several interesting discoveries along the way. As discussed earlier, the integration of message passing and shared memory was one of the most important ones. Another was the discovery of the "window of vulnerability" problem. Although we had known that several livelock scenarios and some deadlock scenarios existed in the presence of multi-phase memory transactions, we had never encountered them in our initial simulations. Soon enough, however, we ran into the first of these, namely a livelock scenario that arises when both an instruction and its associated data item map to the same cache line. Our initial solution involved additional state in the cache tags to recognize and correct this problem by temporarily locking down cache lines. However, as the implementation progressed, it became clear that locking down cache lines introduced deadlocks in the presence of message-passing and LimitLESS traps. Naturally, we had to make the system work, and shortly thereafter we developed an algorithm called "associative thrashlock" and a unified hardware framework called the transaction buffer for solving these problems in general.[6]

The transaction buffer mechanism helped solve yet another problem that we encountered, or more truthfully, took upon ourselves. Late in the design phase, we chose to modify our coherence protocols to support misordering of messages in the network. Our reason for doing this was to develop a more generally applicable solution than that required for our own network (which did deliver messages in order). It turned out that minor modifications to the protocols in concert with the transaction buffer mechanism enabled us to make this significant improvement to our protocols. We also took advantage of the reordering protocols to create a software-based network overflow solution, thereby eliminating the need for multiple networks or virtual channels.

As the implementation progressed, a large body of software was written to make the machine usable. New synchronization and scheduling algorithms were among these (see Lim's and Nussbaum's PhD theses). Interestingly, some discoveries came about through limitations in the prototype. Alewife did not support virtual memory, since we believed we could answer our research questions without it[7]. Barua and Kranz developed a software method called software address translation in which the compiler inlined customizable translations into the code. This method did not see much use when first developed, but we believe that its flexibility combined with the emergence of user-customizable operating systems like Exokernel and SPIN will make it appealing in the future as a replacement or an adjunct to hardware TLBs.

6. Chaiken's Master's thesis discusses the thrashing problems and one of our early solutions. Kubiatowicz's PhD thesis elaborates on the window of vulnerability and the transaction buffer solution. Kubiatowicz's thesis also contains a nice discussion of the chip design, implementation and test efforts, and the futility of chip refabrication in a university.
7. Ken Mackenzie and others built the Fugu system a few years later to explore the issues of protection and virtual memory.

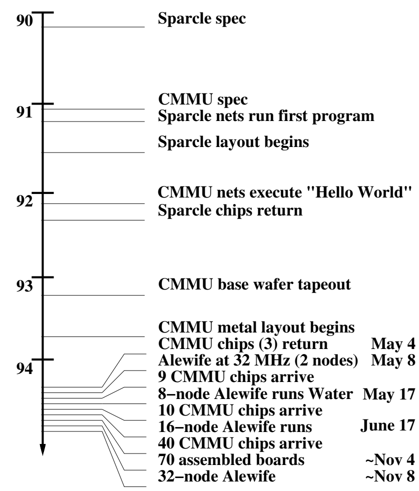
Our collaboration with LSI on the Sparcle processor worked out very well. There was a scary period, though, when Gene Hill left LSI, and the question of why Sparcle was being supported arose. With help from our technical counterparts at LSI and Sun, Godfrey D'Souza at LSI and Mike Parkin at Sun, we were able to obtain the support of Amnon Fisher at LSI, and our Sparcle efforts continued smoothly. As depicted in the timeline in the figure, working Sparcle chips arrived from LSI in early 1992.

We also initiated a collaboration on packaging with Bob Parker, Jeff LaCoss, and Diane Delute at the advanced packaging technology (APT) group at ISI in California. Inspired by Tom Knight's work on pressure connectors, we came up with a prototype design with Jeff LaCoss. This "cool" design had two key features: it did not involve backplane boards, and it was highly compact. Its biggest problem was that the replacement of a single board involved decompressing all the connectors, thereby violating the basic tenet of successful system building — if it ain't broke, don't touch (sic) it. Fortunately, we jettisoned this packaging technology for much less risky technology: passive backplanes, node boards, and traditional connectors.

We first designed and built a node board at MIT. This design was built with a large number of probe points and optimized for table-top debugging. Then, APT turned this prototype board design into a more compact production quality design and also designed the backplane boards. They also came up with the power distribution and cooling design. MIT added the designs for clock distribution and JTAG support.

The following figure shows the progress of our implementation effort. Perhaps the most significant feature of our schedule was our gross underestimation of the time it would take to test the CMMU chip. Fortunately, we got working parts back, and our aggressive software effort that ran concurrently with the chip design paid off handsomely. We had 2 nodes working together in a week, 8 nodes in two weeks, and 16 nodes in a month and a half.

As the machine came online, our evaluation process began. The effort got a massive infusion of enthusiasm and energy when Ricardo Bianchini came to work with us over the summer. Following Ricardo's energetic efforts, we had a large number of the Splash applications ported to Alewife within months. The results from our evaluations are reported in the Alewife paper included in this issue. As we conclude in the Alewife paper, the basic shared memory programming abstraction augmented with mechanisms such as explicit messaging, fine-grain synchronization and context switching, provided both use-performance (ease of use and reasonably good performance) and means for further tuning.



Looking Back

Although it is a little early to look back and assess a project that came to fruition four years ago, it is instructive to do so nonetheless. In particular, it is useful to discuss features that did not deliver on their promise. As might be expected, we grew to love all of Alewife mechanisms, despite all their warts, so it is always difficult to knock any one of them. However, the reader can take a less than enthusiastic response about a feature as a sign of a negative result. Furthermore, as discussed previously and further on in this writeup, many of the features turned out to be useful in unanticipated ways, so the negative results really are often in the context of the anticipated uses.

Perhaps, most importantly, the answers to the two key questions we set out to answer at the initiation of the project, namely, how to exploit locality in a mesh network and how to build a truly scalable directory system, turned out to be of lesser importance than some of the other contributions of the project. The software approach to directories and the integration of message passing and shared memory turned out to have the bigger impact. We further observe that although software-based directories and integrated messaging seem to have impacted other research projects, this impact has not been felt in industry at this time.

It is interesting to speculate on why the issues of scalability we thought so important ten years ago turned out not to be so significant. With the enabling technologies of shared memory programming and cost-effective mesh networks open to scalable machines, we believe their usability and bill-of-materials cost are no longer the issue. We speculate that the reason lies in the nonexistence (at least at this time) of a large class of problems that demand scalable machine performance. Consequently, only a relatively small percentage of the computing world really cares about large-scale multiprocessors. As in the past, a select cadre of users — who are willing to hand-tune their applications — expend extraordinary effort to meet their computational needs. The business case for addressing the needs of this relatively small market remains as elusive today as it was a decade ago.

There are other practical factors that relate to the scalability and applicability of LimitLESS directories in either their hardware-software form or in a purely software form. Although our results demonstrated the compelling cost-performance of two-pointer directories by balancing the hardware and software components of a multiprocessor, we doubt there will ever be commercial or research machines that combine hardware and software like LimitLESS. Since Alewife was an experimental machine, it made sense to implement a few pointers in hardware since we could explore the degradation suffered in going from several pointers to zero pointers. As our results indicated, the five pointer case was competitive with an all-hardware system, and the all-software case was between a factor of two or three worse.

In the commercial environment present at the time of this writing, it makes sense to build all-hardware systems since the hardware overhead is not significant for systems with few tens of processors, and since this approach does not involve modifying existing processor interfaces. Furthermore, multigrain systems allow modest-sized systems to be built by composing smaller machines using software page-based coherence between the components. For these relatively small systems, we will likely see a transition from the hardware approach to a software approach (probably using a separate protocol processor in the short term and a unified processor in the longer term) as the increasing latency gap between the processor clock and main memory makes software attractive.

If Moore's Law ever breaks down, however, scalability will be applicable to mainstream computing (as opposed to marketing hype), and it is very possible that the same market forces might dictate a different tradeoff between software and hardware. Assuming that a new computing paradigm does not emerge, LimitLESS-style coherence could well become appealing for its scaling properties.

One of the surprises after Alewife was built was that many applications written using shared-memory were found to be competitive with the same applications written using message passing. While this was a significant and unanticipated result in favor of shared memory, it was counter to our intuition. A sensitivity analysis revealed later that the state of technology plays a major role in determining which is better. It turns out that shared memory is competitive or better than message passing when the processors are slow compared to the interconnect, and the opposite is true when the processor clocks increase relative to network speeds. We further observed that asynchronous message notification is inherently better suited to operating system-like applications such as CRL.

One of the questions we asked ourselves was whether building was necessary in the face of our sophisticated simulation technology. Not surprisingly, for the applications that ran on our simulators, our results from the real prototype were not qualitatively different from our detailed simulators. However, the availability of the real prototype allowed us to develop a large number of applications and obtain results for realistic problem sizes rapidly. Since there were no major surprises, running these applications and large problem sizes served to validate our conclusions.

And of course, as discussed earlier, many of the key Alewife mechanisms would not have been invented otherwise. Taking the example of integrated messaging, we doubt Kubiatowicz would have even contemplated the introduction of a real message-based I/O mechanism for a simulator. Finally, the simulators could not have reached their level of sophistication had we not been on an implementation path. As a case in point, NWO leveraged many of the same control state machine specifications used in the real hardware. As one of us is fond of saying, if there had been a way to hypnotize ourselves into believing that we were working on a real machine, we could have saved a year spent in design verification whose major value could be measured not in terms of the contributions to science but in value to the soul.

What of fine-grain synchronization and context switching? The insignificant value of hardware support for fine-grain synchronization was one of the salient negative results from the project. As reported by Donald Yeung in his Master's thesis, the means for expressing fine-grain parallelism in the source language is of considerable importance, while the special hardware support for full-empty bits is of marginal value.

The jury on context switching is still out. Context switching is intended to improve the performance of applications with a lot of parallelism that suffer low processor utilization due to their poor cache behavior. Context switching is most useful when the network has a large latency but can deliver high bandwidth. Context switching delivered on its promise for applications with poor memory performance such as MP3D. A dedicated context for handling asynchronous message interrupts without disrupting the computational state on the processor was also valuable. However our applications exhibited reasonable cache behavior, and therefore a reasonable processor utilization. Clearly, MP3D is an exception.[8]

The open question, then, is whether there will be sufficient applications that exhibit poor cache behavior when written in a natural manner under shared memory by average programmers. Looking back, although Bianchini and Lim have done some follow up evaluation of context switching on Alewife,[9] we have been remiss in not expending the effort to find more applications and fully evaluating context switching.

Alewife leveraged many of the advances of previous research such as wormhole-routed low-dimension interconnects, directory based coherence, and efficient message interfaces. In turn, in advancing the notions of software-based directories and integrating messaging and shared memory, we hope it contributed in modest measure to this cycle of research.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280-289 June 1988.

8. One might be tempted to speculate that many of these applications were developed for DASH or Alewife, both cache-based machines, and therefore coded in a cache-friendly style.

9. Bianchini and Lim evaluated context switching on Alewife and published their findings in the August 1996 issue of JPDC. They conclude that "prefetching is preferable over multithreading for machines with low remote access latencies and/or applications with poor locality and consequently short run-lengths. The performance of both techniques is comparable for applications with high remote access latencies and/or good locality." They also argue that context switching has added value in microkernel environments.

Retrospective:

Multiscalar Processors

Gurindar Sohi

Computer Sciences Department
University of Wisconsin
sohi@cs.wisc.edu

Background

We started thinking about the basic multiscalar ideas in about 1988-89: we had been studying out-of-order superscalar processors based upon our RUU design (see paper in this collection) during 1986-1989 [1], and our experiments had given us a lot of insight into the operation of such a machine. We started looking for ways in which we could "simplify" the RUU mechanism. Our main point of attack was the logic needed to implement the instruction scheduling and wakeup functions: we felt that a large centralized instruction window was not a long-term solution.

In October 1989, IEEE Spectrum published the "Microprocessors circa 2000" article [2], with projections of a 100 million transistors on a chip. This article played a very important role in focusing our thoughts. We asked ourselves: how might we use these resources to speed up computation? What would be the architectural paradigm for such a chip? The Intel proposal was what amounted to a 4-way multiprocessor on a chip. We did not feel comfortable with this proposal, since it appeared unlikely that parallelizing compiler technology would be able to automatically parallelize a majority of applications in the foreseeable future.

Developing the ideas

We started out by defining what we considered to be desirable attributes of a circa 2000 microprocessor. These included: (i) easy hardware growth path from one generation to the next, (ii) easy software growth path from one generation to the next, (iii) ability to extract high levels of ILP, across a wide range of applications, with clock speeds comparable to "simple" processors of that era. This implied the need for a microarchitecture built from replicated components, without any (high-utilization) centralized resources that could become a bottleneck from one generation to the next.

We started our search for a circa 2000 paradigm by looking at the dataflow model. We really liked the concepts — thinking about the RUU-based superscalar processor as a dataflow engine had allowed us to get good insight into its operation. However, we were not willing to adopt the model in its entirety. In particular, we were not willing to give up sequential programming semantics, since it appeared unlikely that inherently parallel languages were going to be adopted widely any time soon. This meant that we would have to achieve a dataflow-like execution for a serial program. Rather than consider this a drawback, we considered this an asset: we felt that the inherent sequentiality could be exploited to create localities in the inter-operation communication which could be exploited to decentralize the inter-operation communication mechanism (aka token store in a dataflow machine).

In our experiments with the RUU, we also realized that, though increasing the RUU size would allow more parallelism to be exploited, much of the parallelism was coming from points that were "far apart" in the RUU — there was little parallelism from "close by". Since increasing the RUU size entailed significant overheads, that were not "scalable", we felt that the localities of communication could be exploited to come up with decentralized RUU designs.

We also looked very seriously at the VLIW model. We were quite intrigued by all the parallelism-enhancing transformations that were developed by the VLIW community, but were very uncomfortable with the execution model: the basic model did not expose and exploit localities of communication that we felt were very important to exploit. Moreover, it suffered from the "software problem" in going from one generation to the next. A partitioned VLIW would allow localities of communication to be exploited, but that would mean dividing the parallelism into dependent operation slices (i.e., "vertical slices" vs. "horizontal slices"), and if that had to be done eventually, what was to be gained by packing the operations into a horizontal slice? Why not pack them into vertical slices?

At about the same time Jim Smith introduced me to an interesting microarchitecture he had become aware of several years before. He referred to it as a "dependence architecture", and it was based on an early, never-completed version of the Cray-2. This machine consisted of 8 independent units, each with an accumulator, and collectively backed by a shared register file. Sequences of dependent operations were submitted to each unit, where they would execute in parallel.

The above led us to conceive an architecture in which the instruction window (aka RUU) could be split, and all important aspects of the machine could be decentralized. We considered four important aspects: instruction supply, instruction scheduling, inter-operation communication, and memory data supply. Our experiments with register traffic [4] convinced us that splitting the instruction window would also allow us to decentralize the inter-operation communication and the instruction scheduling. The next issue was that of instruction supply: how should these sub-windows be fed? To decentralize instruction supply, it made sense to start filling the sub-windows from different points in the instruction stream, and so we proposed multiple sequencers to do this. And since the different windows would be operating independently, loads would need to execute before the identities of prior stores (in a different window) were known. This would require a significant rethinking of how memory operations are to be carried out, and we did not arrive at a solution for this problem (the Address Resolution Buffer, or ARB) until the Summer/Fall of 1991.

Manoj Franklin started working on the multiscalar ideas in Fall 1990, and by Spring 1991 we had a simulator to test out the basic concepts.

In May 1991, I gave a talk about the multiscalar ideas at Cray Research, and in June 1991, I gave a talk at DEC, Marlboro. After the talk, I had a long conversation with Joel Emer and Bob Nix (and one other person whose name I can't remember), about the memory system aspects of such a machine. They told me that they had a solution, in the context of a VLIW processor, but were unable to give me details. This led us to think more about the issue, and come up with the ARB in Summer/Fall 1991. (Later, it turned out that the solutions, and the problems they were solving, were entirely different, but there is no doubt in my mind that knowing a solution was possible helped us find one.)

Very early on we realized the need to have code blocks (or ergons as Dionisios Pnevmatikatos and T. N. Vijaykumar called them) that were greater than a basic block. This would require novel prediction techniques that were able to go beyond multiple branches simultaneously, as well as the ability for the machine to resolve multiple branches simultaneously. For this, Pnevmatikatos and Franklin developed the concept of control flow prediction [5].

In Fall 1991, Franklin built a complete simulator, including an ARB, and we submitted a paper to ISCA92 [3], which was accepted. I had also visited several companies to give talks, and had discussed our ideas with many people. In particular, I had detailed discussions with, and received critiques from Mitch Alsup, Jim Smith, and Bob Rau. These discussion were crucial in the development and refinement of the ideas.

In January 1992 I gave the first public presentation at HICSS, at a session organized by Wen-Mei Hwu. I got a lot of hard questions from the small audience, which included Mike Flynn, Andy Heller, Peter Hsu, Wen-Mei Hwu, Yale Patt, and Bob Rau.

In the summer of 1992, Mark Hill convinced us to come up with a name for the concept; the term "Expandable Split Window" was not sufficiently catchy. After trying several variations of scalar, I came up with the name "Multiscalar".

Franklin continued with experiments of the concept in 1991-93. While we had a simulator, we had no compiler: Franklin's simulator collected all the information needed for multiscalar-execution from an existing MIPS binary. Perhaps the first performance impediment that we faced was squashes dues to memory data dependences: in many cases the MIPS compiler would spill a register (assuming it would be a cache hit) and reload it shortly afterwards — this would cause data dependence squashes. Franklin tried to alleviate this problem by implementing selective squashing: only the offending load and its dependent slice of instructions would be restarted. However, this required all the instructions of the task to be in the processing unit. This, and other restrictions placed artificial constraints on task selection: code was divided at arbitrary points, for example, after half the load of a double-word load or after a lui, half-way through building an address (this aggravated data dependences). Moreover, tasks could not be large since loops and function calls could not be included in a task. Overall, we felt comfortable with the basic ideas — Franklin wrote his Ph.D dissertation on it in September 1993, but we felt that we needed to rethink our decisions assuming we could get help from software (many of the problems we were facing with hardware task selection were easily solved with software task selection) as well as from hardware.

In 1992 Scott Breach and T. N. Vijaykumar (or Vijay) started working with me and we started thinking about how we could implement the concepts better than we had. In particular, we wanted to see if we could get high levels of ILP with simple processing cores (which were more likely than sophisticated cores to meet our objective of "comparable" clock), even though with simple cores we might have to sacrifice other performance aspects (for example, the ability to selectively squash or restart instructions). And we wanted to see the power of the multiscalar concept if we had flexibility in both the software and the hardware.

Scott and Vijay jointly took on the responsibility of defining the Multiscalar architecture; Scott took responsibility for the simulator and Vijay the compiler. In 1993-94, we defined a multiscalar architecture, built a simulator and compilation infrastructure (compiler, assembler, linker, disassembler) based upon the GNU tools, and wrote the ISCA95 paper. This was truly an enormous task: we had neither a compiler that could generate a multiscalar binary, nor a simulator that would execute one. Both were developed simultaneously, and when something went wrong, it wasn't clear whether the bug was in the compiler or in the simulator. It took many hours of painstaking effort to get the tools to a point where we could compile and simulate arbitrary C programs and get reliable performance numbers out of the simulations.

When we were writing the ISCA '95 paper, the compiler and simulator were still in their infancy, and we were just starting the process of analyzing and understanding the results. While we were getting impressive results for some benchmarks (at least as compared to other proposals of that time), the extra instructions that the compiler generated for the gcc benchmark, and the microarchitectural parameters chosen, caused a slowdown. We were sorely tempted to remove this result from the paper, with an excuse that our compiler was still unable to compile and generate good code for gcc (a problem that many other research compilers shared). I am really glad we did not. We had always tried to be conservative in our choice of parameters (and the resulting performance numbers) in our simulations and publishing a negative result underlined this philosophy.

An important concept that we discovered after the paper was published (in the 1995-96 time frame) was the notion of data dependence prediction and synchronization. Since our decision to do away with selective squashing ability in the processing cores, performance loss due to data dependence squashes had become an important component of the total performance loss. Andreas Moshovos (along with Scott and Vijay) came up with the idea of predicting if a dependence violation was going to occur, and synchronizing the offending operations [6]. This greatly improved performance for many benchmarks.

The Kestrel Project

In Fall 1994, Jim Smith returned to Wisconsin. He became excited about the concept, and we approached the NSF Experimental Systems Program and DARPA for funding to test out the feasibility and practicality of the concept. We were funded, and we initiated the Kestrel project. The size of the research group grew, with students exploring all aspects of the problem: compiler, simulator, a Verilog implementation (and synthesis) of a sample multiscalar configuration, as well as exploring microarchitectural components of a multiscalar processor, and thinking of alternate implementations. The project continues at the time this retrospective was written. To date we have learned that there are no technical barriers to the implementation of the multiscalar paradigm.

People who have contributed to various aspects of the Kestrel project include: Scott Breach, T. N. Vijaykumar, Andreas Moshovos, Eric Rotenberg, Quinn Jacobson, Jeremy Williamson, Paul Thayer, Selim Bilgin, Matt Kupperman, Subramanya Sastry, Amir Roth, Sridhar Gopal, Matt Mergener, Craig Zilles, Atsushi Okamura, Anand Kamannavar and Padmaja Nandula.

Summary

In 1998, almost 10 years after we first started thinking about the basic multiscalar concepts, we continue to study, develop, and refine them, as we continue our research on microprocessors of the next millennium. Our experience to date with the concepts suggests they are quite promising indeed. We continue to work on related issues at Wisconsin. It is also very exciting to see several related research projects starting out elsewhere.

Acknowledgments

I would like to thank Mike Foster and Zeke Zalcstein of NSF, Gary Koob of ONR (now of DARPA), Bob Parker and Bob Lucas of DARPA, and Konrad Lai of Intel for funding our work in multiscalar processors.

References and Related Papers

[1] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors," *Proc. 14th International Symposium on Computer Architecture, pp.* 27-34, June 1987.

[2] P. P. Gelsinger, P. A. Gargini, G. H. Parker, A. Y. C. Yu, "Microprocessors circa 2000," *IEEE Spectrum*, vol. 26, no. 10, pp. 43-47, October 1989.

[3] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", *Proc. 19th International Symposium on Computer Architecture*, pp. 58-67, May 1992.

[4] M. Franklin and G. S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proc. 25th Annual International Symposium on Microarchitecture (MICRO-25*), December 1992.

[5] D. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control Flow Prediction for Dynamic ILP Processors," *26th Annual International Symposium on Microarchitecture (MICRO-26)*, December 1993.

[6] A. Moshovos, S. E. Breach, T. N. Vijaykumar and G. S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th International Symposium on Computer Architecture*, pp. 181-193, June 1997.

[7] Tom Knight, "An Architecture for Mostly Functional Languages," *Proc. 1986 ACM Symposium of LISP and Functional Programming*, pp. 105-112, August 1986.

[8] Robert A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture", *Proc. 15th l International Symposium on Computer Architecture,* pp. *131-140*, May 1988.

[9] D. E. Culler and Arvind, "Resource Requirements of Dataflow Programs," *Proc. 15th International Symposium on Computer Architecture*, pp. 141-150, May 1988.

[10] Gregory M. Papadopoulos and David E. Culler, "Monsoon: an explicit token-store architecture," *Proc. 17th Annual International Symposium on Computer Architecture*, 1990, pp. 82-91.

Retrospective:

Simultaneous Multithreading: Maximizing On-Chip Parallelism

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
{eggers,levy}@cs.washington.edu
tullsen@ucsd.edu

This paper was published in 1995 and so it seems early to do a retrospective; in fact, research in simultaneous multithreading (SMT) is still ongoing. The original project began in early 1994. We were beginning to see commercial microprocessors that could issue many instructions per cycle (wide superscalars), but which rarely did so due to dependencies and long memory latencies. In fact, processor utilization seemed to be declining as fast as instruction issue width was increasing.

We actually began by looking at targeted solutions to the low processor utilization problem, such as improved branch prediction, but quickly realized that no single such mechanism was likely to solve the overall problem we faced. The graph in the paper attributing the many causes of lost cycles was one key to our intuition, and made us realize that we needed a more global, latency-tolerant solution. This led us to the basic idea of using a much finer-grained multithreading than had been previously attempted as a general way to tolerate *all* forms of lost utilization. The idea was surely influenced by previous designs such as the Tera, MIT Alewife, and M-machine projects, and by Radhika Thekkath's UW thesis. Several other projects had also looked at various forms of multithread, superscalar issue. However, as we examined them, each of these studies seemed to be limited in some way by the constraints of a particular hardware architecture in which it was embedded. None of the previous projects, to our mind, had really explored or analyzed the total potential of the fully-general concept we were considering, nor had they described it in the way we were thinking about it. We chose the name "simultaneous multithreading" to give this general concept a new label.

The more we thought about it, though, the more we realized that simultaneous multithreading (SMT) was significantly different from the traditional (context-switching) multithreading designs. In particular, there was something aesthetically pleasing about the concept of sharing *all* processor resources *every* cycle: basically, just throw all the threads in the machine, and let it make the best dynamic decision about what instructions to send to what functional units at every instant. The result of this was effectively to use thread-level parallelism to make up for a lack of instruction-level parallelism in individual programs. This is a somewhat different goal than that addressed by previous multithreaded designs. If you have one or a few threads with moderate ILP each, that's fine; many threads with a little ILP each, that's fine; multiprogramming, that's fine. The figure that appears in the paper, showing the effect of "horizontal waste" and "vertical waste," was also a useful tool for us in understanding and explaining why SMT was likely to work better than the alternative schemes (superscalar and traditional multithreading, and later single-chip multiprocessors).

There were a few major goals of the work from the start, both arising from our desire to target mainstream processor designs. First, we were very aware that poor single-thread performance would not be acceptable in this market (as opposed to the types of uses the Tera is targeting, for example); therefore, it was crucial that single-thread performance not be harmed by the addition of SMT. Second, we wanted SMT to be easily implementable on state-of-the-art microprocessors. The "state-of-the-art" that eventually facilitated meeting this goal was dynamic instruction issue (i.e., out-of-order processors); in fact, we were a little ahead of this at the time, which caused many people to doubt that SMT was achievable. Following this paper, we were lucky to work with colleagues Joel

Emer and Rebecca Stamm from Digital's Alpha group, who greatly contributed to the microarchitecture design and helped us to show how SMT required only limited changes to an out-of-order processor; we also discovered how SMT performance could be improved significantly by fetching from the “right” threads, i.e., those making best use of the processor. By the time our second paper was published at the following ISCA (1996), many people saw the appearance of out-of-order machines and realized that once you have dynamic instruction issue, you've already provided most of the complexity with respect to the instruction issue mechanism required by SMT. It was quite interesting (and exciting) in retrospect to see a major change in response to the idea of SMT that occurred over the period of less than one year.

Acknowledgments

We would like to thank Digital Equipment Corporation and Equator Technologies Inc. for access to the Multiflow compiler. This work was supported by the National Science Foundation Grant No. MIP-9632977.

About the authors

Susan Eggers is Associate Professor of Computer Science and Engineering at University of Washington. Her research includes computer systems architecture, machine-dependent compiler optimizations, and dynamic compilation techniques.

Henry Levy is Professor of Computer Science and Engineering at the University of Washington. His research focuses on operating system design, computer architecture, and their interaction.

Dean Tullsen finished his PhD at University of Washington on the topic of Simultaneous Multithreading. He is currently Assistant Professor of Computer Science at University of California, San Diego, where he works on architecture and simultaneous multithreading.