Common Bonds: MIPS, HPS, Two-Level Branch Prediction, and Compressed Code RISC Processor

Onur Mutlu
ETH Zurich

Rich Belgard

We are continuing our series of retrospectives for the 10 papers that received the first set of MICRO Test of Time ("ToT") Awards in December 2014.[1,2] This issue features four retrospectives written for six of the award-winning papers. We briefly introduce these papers and retrospectives and hope that you will enjoy reading them as much as we have. If anything ties these works together, it is the innovation they delivered by taking a strong position in the RISC/CISC debates of their decade. We hope the *IEEE Micro* audience, especially younger generations, will find the historical perspective provided by these retrospectives invaluable.

MIPS

The first retrospective is for the oldest paper being discussed in this issue: "MIPS: A Microprocessor Architecture."[3] This work introduced MIPS (Microprocessor without Interlocked Pipeline Stages) and its design philosophy, principles, hardware implementation, related systems, and software issues. It is a concise overview of the MIPS design, which is one of the early reduced, or simple, instruction sets that have significantly impacted the microprocessor industry as well as computer architecture research and education for decades (and probably counting!). The key design principle is to push the burden of performance optimization on the compiler and keep the hardware design simple. The compiler is responsible for generating and scheduling simple instructions, which require little translation in hardware to generate control signals to control the datapath components, which in turn keeps the hardware design simple. Thus, the instructions and hardware both remain simple, whereas the compiler becomes much more important (and likely complex) because it must schedule instructions well to ensure correct and high-performance use of a simple pipeline. Many modern processors continue to take advantage of some of the design principles outlined in this paper, either in their ISA (the MIPS ISA still has considerable use) or in their internal execution of instructions (many complex instructions are broken into simple RISC-style microinstructions in modern processors). The MIPS ISA and design is also used in many modern computer architecture courses because of its simplicity.

Thomas Gross, Norman Jouppi, John Hennessy, Steven Przybylski, and Chris Rowen provide in their retrospective a historical perspective on the development of MIPS, the context in which they arrived at the design principles, and the developments that occurred after the paper. They also provide their reflections on what they see as the design and evaluation decisions made in the MIPS project that passed the test of time. The retrospective touches on the design tradeoffs made to couple the hardware and the software, the MIPS project's effect on the later development of "fabless" semiconductor companies, and the use of benchmarks as a method for evaluating end-to-end performance of a system as, among others, contributions of the MIPS project that have stood the test of time.

High-Performance Systems

The second retrospective addresses three related papers that received MICRO ToT Awards in 2014. These papers introduce the High Performance Substrate (HPS) microarchitecture, examine its critical design issues, and discuss the use of large hardware-supported atomic units to enhance performance in HPS-style, dynamically scheduled microarchitectures.

The first paper, "HPS, A New Microarchitecture: Rationale and Introduction,"[4] introduced the HPS microarchitecture, its design principles, and a prototype implementation. It also introduced the concept of restricted dataflow and its use in implementing out-of-order instruction scheduling and execution while maintaining sequential execution and precise exceptions from the programmer's perspective (that is, without exposing the out-of-orderness of dataflow execution to software).

Published by the IEEE Computer Society

This concept of out-of-order execution using a restricted dataflow engine that supports precise exceptions has been the hallmark of high-performance processor designs since the success of the Intel Pentium Pro,[5] which implemented the concept. The paper describes how complex instructions can be translated in hardware to simple micro-operations and how dependent micro-operations are linked in hardware such that they are executed in a dataflow manner, in which a micro-operation "fires" when its input operands are ready. The mechanisms described in this work paved the way for many future and current processor designs with complex instruction sets to execute programs using out-of-order execution (and exploit some of the RISC principles) without requiring changes to the instruction set architecture or the software. The retrospective by Yale Patt and his former students Wen-mei Hwu, Steve Melvin, and Mike Shebanow beautifully describes the six major contributions of the original HPS paper.

The second paper, "Critical Issues Regarding HPS, A High Performance Microarchitecture,"[6] discusses what the authors deem as critical issues in the design of the HPS microarchitecture. The paper is forward looking, discussing some of the problems that the authors thought "would have to be solved to make HPS viable," as the retrospective puts it. The paper can be seen as a "problem book" that anticipates the many issues to be solved in designing a modern out-of-order execution engine that translates a complex instruction set to a simple set of micro-operations internally. For example, it introduces the notion of a node cache, an early precursor to a modern trace cache, and discusses issues related to control flow, design of the dynamic instruction scheduler, the memory units, and state repair mechanisms. It also introduces the *unknown address problem*—the problem of whether a younger load instruction can execute in the presence of an older store instruction whose address has not yet been computed. This problem, now more commonly known as the *memory disambiguation problem,* has since been the subject of many works.

The third paper, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines,"[7] likely provides the first treatment of large units of work as atomic units that can enable higher performance in an HPS-like dynamically scheduled processor. It discusses the benefits of enlarged blocks in an out-of-order microarchitecture and describes how enlarged blocks can be formed at different layers: architectural, compiler, and hardware. It introduces the notion of a fill unit, which is a hardware structure that enables the formation of large blocks of micro-operations that can be executed atomically. This work has opened up new dimensions and is considered a direct precursor to notions such as the trace cache, which is implemented in the Intel Pentium 4,[8] and the block-structured ISA.[9]

In their retrospective, Yale Patt and colleagues provide a historical perspective of HPS, the environment and the works that influenced the design decisions behind it, its reception by the community and the industry, and what happened afterward. We hope you enjoy reading this retrospective that describes the inspirations and development of a research project that has heavily influenced almost all modern high-performance microprocessor designs, CISC or RISC. The authors also provide a glimpse of some of the future research that has been enabled and influenced by the HPS project, of which the next paper we discuss is a prime example.

Two-Level Branch Prediction

The third retrospective is for "Two-Level Adaptive Training Branch Prediction"[10] by Tse-Yu Yeh and Yale Patt. This paper addresses a critical problem in the HPS microarchitecture, and in pipelined and out-of-order microarchitectures in general: how to feed the pipeline with useful instructions in the presence of conditional branches. The authors introduced a breakthrough design for conditional branch prediction, which they dubbed the *two-level branch predictor.* This predictor is based on the realization that adding another level of history to predict a branch's direction can greatly improve branch prediction accuracy, in addition to the single level that tracks which direction the same static branch went the last *N* times it was executed (which had been established practice since Jim Smith's seminal ISCA 1981 paper[11]). In other words, the direction the branch took the last *N* times the same "history of branch directions" was encountered can provide a much better prediction for where the branch might go the next time the same "history of branch directions" is encountered.

This insight formed the basis of many future branch predictor designs used in high-performance processors, most notably starting with the Pentium Pro. Yeh and Patt's MICRO 1991 paper discusses various design choices for this predictor and provides a detailed evaluation of their impact on prediction accuracy. For example, they examine the effect of keeping global or local branch history registers (many modern hybrid predictors use a combination) and the effect of how the pattern history tables are organized. The retrospective provides the historical perspective as to how two-level branch prediction came about, along with the contemporary works that followed it.

Compressed Code RISC Processor

The final retrospective in this issue is for "Executing Compressed Programs on an Embedded RISC Architecture,"[12] one of the first papers to tackle the problem of code compression to save physical memory space in cost-sensitive embedded processors. It is also one of the first to examine cost-related architectural issues in a RISC-based system on chip (SoC), as Andy Wolfe's retrospective nicely describes. The basic idea is to store programs in compressed form in main memory to save memory space, and decompress each cache block right before it is fetched into the processor's instruction cache. Wolfe's retrospective shows clearly why RISC made sense for embedded SoC designs, owing to such designs' cost

constraints and real-time requirements. He discusses how he came to work on designing such embedded RISC processors, the challenges associated with such designs, and the series of alternatives he examined to reduce the cost of an embedded RISC processor. He also provides a nice perspective of the requirements, history, and development of the embedded computing market, which we hope you will greatly enjoy reading.

As we conclude our introduction of this set of six MICRO ToT Award winners, we make a few observations. First, as the retrospectives make clear, all these papers have had widespread impact on both modern processor designs as well as academic research, which has made them worthy of the Test of Time Award. Second, each of these papers vigorously challenged the status quo and took contrarian positions to the dominant design paradigms or popular approaches of their times, which is how they achieved their impact. MIPS (RISC) was proposed as a compelling alternative to CISC, which was the dominant ISA design paradigm at the time; HPS was proposed as a way to enable high-performance CISC against the strong advocacy for RISC at the time; two-level branch prediction came as a breakthrough when Jim Smith's decade-old two-bit counters were thought to be the best predictor available; and cost-sensitive embedded RISC processors were proposed as an alternative to simple low-performance microcontrollers in the embedded computing market. Third, and perhaps most importantly, in all these papers, the focus on quantitative evaluation was much less than a typical paper we see published today in top computer architecture conferences. In fact, the three seminal HPS papers have no evaluation at all! As we go through these award-winning papers, we cannot help but ask the question, "Could these award-winning papers with little or no quantitative evaluation have been published today if they were submitted to our top computer architecture conferences?"

We hope our community opens a dialogue to seriously consider this question when evaluating papers for acceptance. Contrarian ideas that change many things in a system might not easily be quantitatively evaluated in a rigorous manner (and they perhaps should not be), but that does not mean that they cannot change the world or that the entire scientific community cannot greatly benefit from their publication and presentation at our top conferences. As the computer architecture field evolves and diversifies in a period where there does not seem to be a dominant architectural design paradigm, it is time to take a step back and enable the unleashing of bigger ideas in our conferences by relaxing the perceived requirements for quantitative evaluation. We hope the MICRO ToT Award winners and their retrospectives provide inspiration and motivation to this end, in addition to inspiring young members of our community to develop ideas that challenge the status quo in major ways.

References

1. O. Mutlu and R. Belgard, "Introducing the MICRO Test of Time Awards: Concept, Process, 2014 Winners, and the Future," *IEEE Micro*, vol. 35, no. 2, 2015, pp. 85–87.
2. O. Mutlu and R. Belgard, "The 2014 MICRO Test of Time Award Winners: From 1978 to 1992," *IEEE Micro*, vol. 36, no. 1, 2016, pp. 60–C3.
3. J. Hennessy et al., "MIPS: A Microprocessor Architecture," *Proc. 15th Ann. Workshop Microprogramming*, 1982, pp. 17–22.
4. Y.N. Patt, W.M. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Ann. Workshop Microprogramming*, 1985, pp. 103–108.
5. D.B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, Apr. 1996, pp. 14–15.
6. Y.N. Patt et al., "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proc. 18th Ann. Workshop Microprogramming*, 1988, pp. 109–116.
7. S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. 21st Ann. Workshop Microprogramming and Microarchitecture*, 1988, pp. 60–63.
8. G. Hinton et al., "The Microarchitecture of the Intel Pentium 4 Processor," *Intel Technology J.*, vol. 1, 2001, pp. 1–12.
9. S.W. Melvin and Y.N. Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA," *Int'l J. Parallel Programming*, vol. 23, no. 3, 1995, pp. 221–243.
10. T.-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61.
11. J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Ann. Symp. Computer Architecture*, 1981, pp. 135–148.
12. A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, 1992, pp. 81–91.

Onur Mutlu is a full professor of computer science at ETH Zurich and a faculty member at Carnegie Mellon University. Contact him at onur.mutlu@inf.ethz.ch.

Rich Belgard is an independent consultant for computer manufacturers, software companies, and investor groups and an expert and consultant to law firms. Contact him at belgard@gmail.com.

A Retrospective on "MIPS: A Microprocessor Architecture"

Thomas R. Gross
ETH Zurich

Norman P. Jouppi
Google

John L. Hennessy
Stanford University

Steven Przybylski
Verdande Group

Chris Rowen
Cadence Design Systems

The MIPS project started in early 1981. At that time, mainstream architectures such as the VAX, IBM 370, Intel 8086, and Motorola 68000 were fairly complex, and their operation was controlled by microcode. Designers of high-performance implementations discovered that pipelining of complex-instruction-set computer (CISC) machines, especially the VAX, was hard; the VAX 11/780 required around 10 processor cycles to execute a single instruction, on average. Tradeoffs that had been made when there were few resources available for implementation (for example, dense instruction sets sequentially decoded over many cycles by microcode) ended up creating unnecessary constraints when more resources became available. For example, as transistors were replacing the core, the cost of memory dropped much faster than logic—favoring a slightly less dense program encoding over logic-intensive interpretation. Also, the design of instruction sets did not assume the use of an optimizing compiler. CISC instruction features such as arithmetic operations with operands in memory were especially difficult to pipeline and did not take advantage of compiler capabilities (such as register allocation) that could provide more efficient execution.

One area of architecture research in the early 1980s was to create even more complex CISC architectures. Notable examples of these include high-level language machines such as Lisp machines, and later the Intel 432. However, the approach taken in the MIPS project as well as the RISC project at the University of California, Berkeley, was to design simpler instruction sets that did not require execution of microcode. These simpler instructions were a better fit for an optimizing compiler's output—more complex operations could be synthesized from several simple operations, but in the common case where only a simple operation was needed, the more complex aspects of a CISC instruction could be effectively optimized away. In the case of the MIPS project, we emphasized ease of pipelining and sophisticated register allocation, whereas the Berkeley RISC project included support for register windows in hardware.

The quarter before the MIPS project started, Stanford University had offered for the second time a (graduate) class on VLSI design, based on the approach developed by Carver Mead and Lynn Conway.[1] One key idea was that the design rules could be specified without close coupling to the details of a specific process. Most of the MIPS project team members had taken this class. Another difference was that in VLSI, tradeoffs differ from those made in multiboard machines built from hundreds of small-, medium-, and large-scale integration parts like the VAX 11/780. In a multiboard machine, gate transistors were expensive (for example, only four 2-input gates per transistor-transistor logic package), whereas ROM transistors were relatively cheap (for example, kilobits of ROM per package). Much of this was driven by pinout and other limitations of DIP packaging. This naturally favored the design of microcoded machines, in which control and sequencing signals could be efficiently stored as many bits per package instead of being computed by low-density gates requiring many packages.

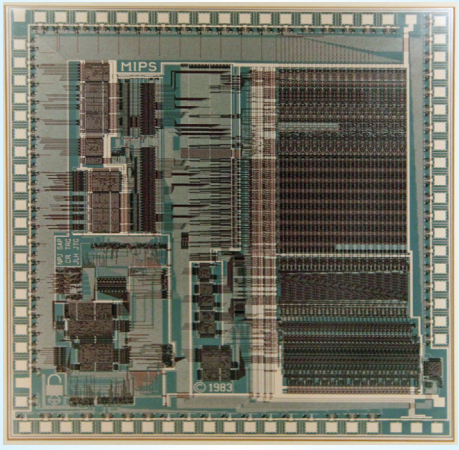

Figure 1. Die photo of MIPS processor.

However, in VLSI, a transistor has approximately the same cost no matter what logical function it is used for, which favors the adoption of direct control as in early RISC architectures instead of microcode. Another difference with widespread adoption of VLSI was that the topology of the wiring was important. In 1981, we had only a single level of metal, so long wires had to be largely planar. This also favored simple and regular designs. Finally, in a VLSI design it was clearly zero-sum: we had a maximum chip size available, so any feature put in would have to justify its value and would displace a less valuable feature. These constraints brought a lot of clarity of focus to the design process.

Given this context, we decided to build a single-chip high-performance microprocessor that could be realized with the fabrication technology available to university teams—a 4-μm single-level metal nMOS process that put severe limitations on the design complexity and size. (The DARPA MOSIS program brokered access to silicon foundries so that students and researchers in universities could obtain real silicon parts.) High performance at that time meant that the processor could execute more than 1 million (meaningful) instructions per second (MIPS), and the design target was 2 MIPS (that is, a clock rate of 4 MHz). Relative to other processor designs of this era, a board powered by a MIPS processor could deliver the same performance as a cabinet-sized computer built with lower levels of integration. At the time we wrote our paper in the fall of 1982,[2] the architecture and microarchitecture were defined, compilers for C and Pascal were written, and several test chips covering different parts of the design had been sent out for fabrication.

Developments After Publication

The MIPS design was completed in the spring of 1983 and sent out for fabrication.[3] At about the time the design was sent for fabrication, the Center for Integrated Systems at Stanford University developed a 3-μm VLSI fabrication process and used the MIPS design (shrunk optically) to tune its process. In early 1984, we received working 3-μm MIPS processors from MOSIS that operated at the speed we had expected for the design at 4 μm. Figure 1 shows the die photo. These processors were run on a test board developed by our fellow grad student, Anant Agarwal, and on 20 February 1984, the first program (8queens) was run to completion. In mid-1984, MIPS Computer Systems was founded. It designed a completely new processor that was heavily influenced by the Stanford MIPS design and was sold as the MIPS R2000. A final evaluation of the Stanford MIPS architecture was published in 1988.[4]

Reflections

Several papers described the implementation and testing of the MIPS processor[5] and discussed the design decisions that turned out to be right and those that deserve to be reconsidered. More than 30 years later, at a time when microprocessors contain multiple independent processing units (cores) and more than 5 billion transistors, a few of these have passed the test of time.

MIPS Design

As part of the MIPS project, we developed various CAD tools, such as tools for programmable logic array synthesis and timing verification.[6] These tools were driven by the designer's requirements and allowed a small team of students and faculty to complete the processor design in a timely manner.

We also invested heavily in compilers beyond the immediate needs of the MIPS processor development. Fellow student Fred Chow designed and implemented a machine-independent global optimizer, and fellow student Peter Steenkiste developed a Lisp compiler to investigate dynamic type checking on a processor that did not provide any dedicated hardware support for this task.

Personal workstations were a novelty at that time, and the MIPS project members were the first to enjoy workstations in their offices. These workstations let us support the implementation with novel tools and interactively experiment with different compiler and hardware optimizations. RISC designs emphasize efficient resource usage and encourage designers to employ resources where they can contribute the most; this RISC strategy drove many of the major design decisions (for example, to abandon microcode in favor of simple instructions, or to use precious on-chip transistors for

frequent operations and leave other operations to be dealt with by software).

Hardware/Software Coupling

The MIPS project invested in software early on; the compiler tool chain worked before the design was finalized. We made several key microarchitecture decisions (including the structure of the pipeline) after running benchmark programs and assessing the impact of proposed design changes.

The MIPS project also emphasized tradeoffs across layers as defined by then-prevalent industry practice. Two sample design decisions illustrate this aspect: the MIPS processor uses word addresses, not byte addresses. The reasoning was that byte addressing would complicate the memory interface and slow down the processor, that bytes aren't accessed that often, and that an optimizing compiler could handle any programming language issues. This decision was probably correct for research processors (and saved us from debating if the processor should be big-endian or little-endian) and illustrates the freedom afforded by looking beyond a single layer. But all subsequent descendants supported byte addressing (of course, by that time, VLSI had advanced to CMOS with at least two levels of metal, so byte selection logic was easier to implement). However, word alignment of word accesses as in MIPS was still retained, unlike in minicomputers.

Sometimes the absence of interlocks on MIPS is seen as a defining feature of this project. However, it was a tradeoff, based on the capabilities of the implementation technology of the time. To meet the design goal of a 4-MHz clock, the designers had to streamline the processor, and still most of the critical paths involved the processor's control component.[7] For the MIPS processor, it made sense to simplify the design as much as possible; later descendants, with access to better VLSI processes, made different decisions. The absence of hardware interlocks (to delay an instruction if one of the operands wasn't ready) was a tradeoff between design complexity, critical path length, and software development costs. In addition to clever circuit design and novel tools, this ability to make hardware/software tradeoffs was a key factor for success.

The team wanted to pick a name for the project that emphasized performance. About nine months earlier, the RISC project at UC Berkeley had started, so we needed a catchy acronym. "Million instructions per second" (MIPS) sounded right, given the project's goals, but this metric was also known as the "meaningless indicator of processor speed." So, we settled on "microprocessor without interlocked pipeline stages."

The Mead/Conway approach to VLSI design emphasized a decoupling of architecture and fabrication. No longer was a chip design tied to a specific (often proprietary, in-house) process. The MIPS project demonstrated that a high-performance design could be realized in this framework and supported the view that VLSI design could be done without close coupling to a proprietary process. Vertical integration—that is, design and fabrication in one company—might offer benefits, but so does the separation of fabrication and design. The MOSIS service was an early (nonprofit) experiment; a few years after the end of the MIPS project, commercial silicon foundries started to offer fabrication services and allowed the creation of "fabless" semiconductor companies.

Benchmarks

The MIPS project used a quantitative approach to decide on various features of the processor and therefore needed a collection of benchmark programs to collect the data needed for decision making. In retrospect, the benchmarks we used were tiny and did not include any significant operating systems code. Consequently, the designers focused on producing a design that delivered performance for compiled programs but paid less attention to the operating system interface and the need to connect the processor to a memory hierarchy. At conferences, benchmark results started to become important, and the MIPS paper (as well as papers by other design groups, such as the UC Berkeley RISC project[8]) presented empirical evidence. Stanford University published the set of benchmarks used by the MIPS project (the Stanford Benchmark Suite), and despite their limitations, they were in use (at least) 25 years later to explore array indexing and recursive function calls. About five years later came the founding of the SPEC consortium, which eventually produced a large body of realistic benchmarks. In 2011, about 30 years later, the first ACM conferences initiated a process that lets authors of accepted papers submit an artifact collection (the benchmarks and tools used to generate any empirical evidence presented in a paper).[9] The MIPS project cannot claim credit for these developments, but it emphasized early on that end-to-end performance, from source program to executing machine instructions, is the metric that matters.

The Stanford MIPS project was an important evolutionary step. Later RISC architectures such as MIPS Inc. and DEC Alpha were able to learn from both our mistakes and successes, producing cleaner and more widely applicable architectures, including integrated floating-point and system support features such as TLBs. All new architectures that appeared afterward (since 1985) incorporated ideas from the RISC designs, and as the concern for resource and power efficiency continues to be important, we expect RISC ideas to remain relevant for processor designs. And, finally, this MIPS paper was also an important step in the transition of the SIGMICRO Annual Workshop on Microprogramming to the IEEE/ACM International Symposium on Microarchitecture.

References

1. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

2. J. Hennessy et al., "MIPS: A Microprocessor Architecture," *Proc. 15th Ann. Microprogramming Workshop*, 1982, pp. 17–22.
3. C. Rowen et al., "MIPS: A High Performance 32-bit NMOS Microprocessor," *Proc. Int'l Solid-State Circuits Conf.*, 1984, pp. 180–181.
4. T.R. Gross et al., "Measurement and Evaluation of the MIPS Architecture and Processor," *ACM Trans. Computer Systems*, Aug. 1988, pp. 229–258.
5. S. Przybylski, "The Design Verification and Testing of MIPS," *Proc. Conf. Advanced Research in VLSI*, 1984, pp. 100–109.
6. N. Jouppi, "TV: An NMOS Timing Analyzer," *Proc. 3rd CalTech Conf. VLSI*, 1983, pp. 71–85.
7. S. Przybylski et al., "Organization and VLSI Implementation of MIPS," *J. VLSI and Computer Systems*, vol. 1, no. 2, 1984, pp. 170–208.
8. D.A. Patterson and C.H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. 8th Ann. Symp. Computer Architecture*, 1981, pp. 443–457.
9. S. Krishnamurthi, "Artifact Evaluation for Software Conferences," *SIGPLAN Notices*, vol. 48, no. 4S, 2013, pp. 17–21.

Thomas R. Gross is a faculty member in the Computer Science Department at ETH Zurich. Contact him at thomas.gross@inf.ethz.ch.

Norman P. Jouppi is a distinguished hardware engineer at Google. Contact him at jouppi@acm.org.

John L. Hennessy is president emeritus of Stanford University. Contact him at hennessy@stanford.edu.

Steven Przybylski is the president and principal consultant of the Verdande Group. Contact him at sp@verdande.com.

Chris Rowen is the CTO of the IP Group at Cadence Design Systems. Contact him at rowenchris@gmail.com.

HPS Papers: A Retrospective

YALE N. PATT
University of Texas at Austin

WEN-MEI W. HWU
University of Illinois at Urbana–Champaign

STEPHEN W. MELVIN

MICHAEL C. SHEBANOW
Samsung

HPS happened at a time (1984) when the computer architecture community was being inundated with the promises of RISC technology. Dave Patterson's RISC at Berkeley and John Hennessy's MIPS at Stanford were visible university research projects. Hewlett Packard had abandoned its previous instruction set architecture (ISA) in favor of the HP Precision Architecture (HP/PA) and had attracted a number of people from IBM to join their workforce. Sun Microsystems had moved from the Motorola 68020 to the Sparc ISA. Motorola itself was focusing on their 78K, later renumbered 88K. Even Intel was hedging its bets, pursuing the development of the i860 along with continued activity on x86.

The RISC phenomenon rejected the VAX and x86 architectures as far too complex. Both architectures had variable-length instruction sets, often with multiple operations in each instruction. The VAX Index instruction, for example, required six operands to assist in computing the memory location of a desired element in a multidimensional subscripted array. If you wanted the location of A[I,J,K], you could obtain it with three instantiations of the Index instruction. The Index instruction took one of the subscripted variables—say I—and checked the upper and lower bounds; if they both passed, it used the size information of each element in the array and the array's dimensions to compute part of the location of A[I,J,K]. More than a half-dozen operations were performed in carrying out the work of this instruction. Intel's x86's variable-length instruction had prefixes to override an instruction's normal activity, and several bytes when necessary to locate the location of an operand.

RISC advocates argued that with simpler instructions requiring in general a single operation, wherein the signals needed to control the datapath were

contained within the instruction itself, one would require no microcode and could make the hardware much simpler, resulting in a program executing in less time. Not everyone followed the mantra exactly; for example, Hennessy's MIPS used a pipeline reorganizer to package two operations in a single instruction. But for the most part, RISC meant single-cycle execution of individual instructions, statically scheduled and fetched one at a time. The instructions were essentially micro-ops.

We marched to a different drum. "We" were Yale Patt, a visiting professor in the fifth year of his nine-year visit at UC Berkeley, and Wen-mei Hwu, Michael Shebanow, and Steve Melvin, all PhD students at Berkeley who completed their PhDs there with Yale over the next several years. We dubbed our project the High Performance Substrate (HPS). We were all part of the Aquarius project, led by Al Despain. Al concentrated on designing a Prolog engine, and we focused on high-performance microarchitecture.

Berkeley was a special place to do research in the 1980s. Among the work going on, Velvel Kahan invented and refined his specification of IEEE floating-point arithmetic; Michael Stonebraker invented Ingres; Domenico Ferrari produced distributed UNIX; Manuel Blum, Dick Karp, and Mike Harrison pursued exciting theoretical issues; Lotfi Zadeh refined his fuzziness concepts; and Sue Graham did important work in compiler technology.

Unlike the RISC projects going on at the time, we felt it was unnecessary to meddle with the ISA; rather, we would require the microarchitecture to break instructions into micro-operations (AMD later called them Rops, for RISC ops) and allow the micro-ops to be scheduled to the function units dynamically (that is, at runtime). We insisted on operating below the ISA's level, allowing the ISA to remain inviolable and thereby retaining all the work of the previous 50 years. We were sensitive to not introduce features that might seem attractive at the moment but could turn out to be costly downstream.

Several pieces of previous work provided insights. Twenty years earlier, in the mid-1960s, the IBM 360/91 introduced the Tomasulo algorithm, a dynamic scheduling (aka out-of-order execution) mechanism, in their FPU.[1] Instructions were allowed to schedule themselves out of program order when all flow dependencies (RAW hazards) were satisfied. Unfortunately, instructions were allowed to retire as soon as they completed execution, breaking the ISA and preventing precise exceptions. In fact, the 360/91 design team had to get special permission from IBM to produce the 360/91 because it did not obey the requirements of the System 360 ISA.[2] IBM did not produce a follow-on product embracing the ideas of the 360/91 FPU.

Ten years earlier, in 1975, Arvind and Kim Gostelow[3] and Jack Dennis and David Misunas[4] introduced the important notion of representing programs as dataflow graphs, and Robert Keller[5] introduced the notion of a window of instructions—that is, instructions that had been fetched and decoded but not yet executed. When we studied these papers in 1984, several things became clear to us.

First, with respect to the Tomasulo algorithm, we would have to do something about the lack of precise exceptions. We correctly decided that this problem was the deal breaker that prevented IBM from producing follow-on designs. We introduced the Result Buffer, which stored the results of instructions executed out of order, allowing them to be retired in order. Although we did not know it at the time, Jim Smith and Andy Pleszkun were concurrently investigating in-order retirement of out-of-order execution machines, and came up with the name Reorder Buffer (ROB), which is a much better descriptor for our Result Buffer.[6]

Second, we recognized that Dennis and Arvind's dataflow graph concept could allow huge gains in performance by allowing many instructions to execute concurrently—and that although it would be near-impossible for a global scheduler to see the available parallelism in an irregularly parallel program, it did not matter. It only mattered that the operations themselves knew when they were ready to execute. By allowing the operations themselves to determine when they were ready to execute, we could exploit irregular parallelism. Allowing the nodes to determine when they can fire is the hallmark of dataflow.

Third, we recognized that, unlike Tomasulo, the mechanism need not be restricted to the FPU—that indeed other arithmetic logic unit operations, and more importantly, loads and stores, could also be allowed to execute in parallel. This let us build the dataflow graph from all instructions in the program, not just the floating-point instructions.

Fourth, we recognized that we did not have to change the ISA and in fact not doing so provided two benefits:

- Our HPS microarchitecture could be used with any ISA. That is, instructions in any ISA could be decoded (that is, converted) to a dataflow graph and merged into the dataflow graph of the previously decoded instructions. The dataflow graph's elements were single operations: micro-ops. Decoding could produce multiple operations each cycle, that is, wide issue.
- Importantly, we could maintain the ISA's inviolability.

Fifth, we recognized that to make this work, we needed to be able to continuously supply a lot of micro-operations into the execution core. We incorporated a post-decode window of instructions, but knew we could not, as Keller advocated, stop supplying micro-ops when we encountered an unresolved conditional branch. For this we would need an aggressive branch predictor, and beyond that, speculative execution. We did not have a suitable branch predictor at the time, but we knew one would be necessary for HPS to be viable. It would also require a fast mechanism for recovering

the state of the machine if a branch prediction took us down the wrong path.

Finally, we recognized that unlike previously constructed dataflow machines, wherein the compiler constructed a generally unwieldly dataflow graph, our dataflow graph would be constructed dynamically. This allowed nodes of the dataflow graph to be created in the front end of the pipeline when the instructions were fetched, decoded, and renamed, and removed from the dataflow graph at the back end of the pipeline when the instructions were retired. The result: at any point in time, only a small subset of the instruction stream, the instructions in flight, are in the dataflow graph. We dubbed this concept "restricted dataflow." We saw that although pure dataflow was problematic for many reasons (such as saving state on an interrupt or exception, debugging programs, and verifying hardware), restricted dataflow was viable.

These revelations did not come all at once. We had the luxury at the time of meeting almost daily in Yale's office as we argued and struggled over the concepts. Sometimes one or more of us were ready to give up, but we didn't.

At the time, many of our colleagues were skeptical of our work and made it clear they thought we were barking up the wrong tree. We were advocating future chips that would use the HPS microarchitecture to fetch and decode four instructions each cycle. Our critics argued that there was not enough parallel work available to support the notion of four-wide issue, and besides, there were not enough transistors on the chip to handle the job. To the first criticism, we pointed out that although we could not see the parallelism, it did not matter. The parallelism was there, and the dataflow nodes would know when all dependencies had been satisfied and they were ready to fire. To the second argument, we begged for patience. Moore's law was alive and well, and although at the time there were fewer than 1 million transistors on each chip, we had faith that Moore's law would satisfy that one for us.

We started publishing the HPS microarchitecture with two papers at MICRO 18. The first paper provided an introduction to the HPS microarchitecture.[7] We described the rationale for a restricted dataflow machine that schedules operations out of order within an active window of the dynamic instruction stream. We showed the need for an aggressive branch predictor that allowed speculative execution, and we provided a mechanism for quickly repairing the state of the machine when a branch misprediction occurs. We introduced the notion of decoding complex instructions into micro-ops, and the use of the result buffer to retire instructions in program order. The second paper at MICRO 18, the "Critical Issues" paper, pointed out some of the problems that would have to be solved to make HPS viable.[8] We discussed various issues, including caching and control flow, and discussed what we called the "unknown address problem." We presented an algorithm for determining whether a memory read operation is ready to be executed in the presence of older memory writes, including those with unknown addresses.

We continued the work over the next several years, publishing results as we went along. In one of those subsequent papers, published at MICRO 21, we introduced the concept of a fill unit and the treatment of large units of work as atomic units.[9] In all, we published more than a dozen papers extending the ideas first put forward in 1985. More importantly, starting with Intel's Pentium Pro, announced in 1995, the microprocessor industry embraced the fundamental concepts of HPS: aggressive branch prediction leading to speculative execution; wide-issue, dynamic scheduling of micro-ops via a restricted dataflow graph; and in-order retirement enabling precise exceptions.

Yale Patt continued as a visiting professor at Berkeley for four more years, then moved on to the University of Michigan and eventually to the University of Texas at Austin. Wen-mei Hwu finished his PhD in 1987 and took a faculty position at the University of Illinois at Urbana–Champaign. Steve Melvin completed his PhD in 1990 and has been a successful independent consultant in the computer architecture industry for more than 25 years. Michael Shebanow completed his PhD and went on to become a lead architect on many microprocessors over the years, including the M88120 at Motorola, the R1 and Denali at HAL, the M3 at Cyrix, and the Fermi GPU shader core complex at Nvidia. He is now a vice president at Samsung, leading a team focused on advanced graphics IP.

References

1. R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, vol. 11, no. 1, 1967, pp. 25–33.
2. R.M. Tomasulo, lecture at University of Michigan, Ann Arbor, Jan. 30, 2008; www.youtube.com/watch?v=S6weTM1tNzQ.
3. Arvind and K.P. Gostelow, *A New Interpreter for Dataflow and Its Implications for Computer Architecture*, tech. report 72, Dept. of Information and Computer Science, Univ. of California, Irvine, 1975.
4. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," *Proc. 2nd Int'l Symp. Computer Architecture*, 1975, pp. 126–132.
5. R.M. Keller, "Look Ahead Processors," *ACM Computing Surveys*, vol. 7, no. 4, 1975, pp. 177–195.
6. J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. IEEE/ACM Int'l Symp. Computer Architecture*, 1985, pp. 36–44.
7. Y.N. Patt, W.M. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Ann. Workshop Microprogramming*, 1985, pp. 103–108.
8. Y.N. Patt et al., "Critical Issues Regarding HPS, A High Performance

Microarchitecture," *Proc. 18th Ann. Workshop Microprogramming*, 1985, pp. 109–116.
9. S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. 21st Ann. Workshop Microprogramming and Microarchitecture*, 1988, pp. 60–63.

Yale N. Patt is the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin. Contact him at pattyn@austin.utexas.edu.

Wen-mei W. Hwu is the AMD Jerry Sanders Endowed Chair in Electrical Engineering at the University of Illinois at Urbana–Champaign. Contact him at w-hwu@uiuc.edu.

Stephen W. Melvin is an independent consultant in the computer architecture industry. Contact him at melvin@zytek.com.

Michael C. Shebanow is a vice president at Samsung. Contact him at shebanow@gmail.com.

Retrospective on "Two-Level Adaptive Training Branch Prediction"

Tse-Yu Yeh
Apple

Yale N. Patt
University of Texas at Austin

Looking back at the world of computer architecture in 1991, when we wrote our MICRO paper,[1] we believe our motivation for addressing branch prediction is best illustrated by our earlier 1991 ISCA paper,[2] which showed that instructions per cycle can be greater than two. Conventional wisdom at the time had argued otherwise. We demonstrated in that paper that there was enough instruction-level parallelism in nonscientific workloads to support our contention. We further showed that the HPS microarchitecture (a superscalar out-of-order execution engine) could improve the execution rate greatly by exploiting instruction-level parallelism, provided the penalty caused by incorrect branch predictions was minimized.

In fact, the lack of an aggressive, highly accurate branch predictor was a major stumbling block in enabling our high-performance microarchitecture to achieve its potential. The HPS microarchitecture required the execution core to be kept fully supplied with micro-ops, organized as a dataflow graph that allowed lots of concurrent execution. This meant a powerful and effective branch predictor that could fetch, decode, and execute instructions speculatively, and only very infrequently have to throw away the speculative work because of a branch misprediction. The computer architecture community had pretty well given up on branch prediction providing any major benefits. Most accepted as fact that Jim Smith's 2-bit saturating counter was as good as it was going to get.[3] His branch predictor was published in ISCA in 1981 and was still the best-performing branch predictor 10 years later. It was the branch predictor in Pentium, Intel's "brainiac" microprocessor, the top of the x86 line at the time, released in 1991.

Tse-Yu spent the summer of 1990 at Motorola, working for Michael Shebanow, one of the original inventors of the HPS microarchitecture. Michael was interested in branch prediction, and in fact almost did his PhD dissertation at Berkeley on branch prediction. Tse-Yu and Michael had many conversations about microarchitecture tradeoffs that summer, and often the discussions would gravitate to branch prediction. When Tse-Yu returned to Ann Arbor for the fall semester, he was excited about the possibility of a branch predictor that could support the HPS microarchitecture.

After many after-midnight meetings over several months in Yale's office, combined with a whole lot of simulation, the two-level adaptive branch predictor was born. We knew the branch predictor would have to be dynamic; that was a no-brainer. The branch predictor would have to use historical information based on the input data that the program was processing, and it would have to accommodate phase changes. But that was not enough. What historical information would yield good branch prediction? Somehow we stumbled on the answer: it was not the direct record of a branch's behavior that was needed, but rather whether the branch was taken at previous instances of time having the same history. That is, if the

branch's history was 011000101 (0 means not taken, 1 means taken), there is little one can infer. But, if the last time the history was 011000101 the branch was taken, and the time before that when the history was 011000101 the branch was also taken, it was a good bet that the branch would be taken at this time.

With that insight, we set out to design the two-level predictor. Several issues had to be resolved.

First, how should the direct record of branches be stored? Three choices presented themselves: a global history register to record the behavior of each dynamic branch; a separate history register for each static branch; and a half-way measure, in which we partitioned branches into equivalence classes and allocated one history register for each class. We dubbed the three schemes G, P, and S, respectively. The obvious tradeoffs were interference versus correlation and the cost of the history registers.

Second, how should the information be stored that records what happens as a result of a previous instance of the current history? We decided to store results as entries in a table, indexed by the bits of history in the history registers. The same three choices applied here—a global pattern table, a per-static-branch pattern table, and the in-between scheme. We dubbed the three choices g, p, and s. Thus, we had nine choices to investigate: GAg, GAs, GAp, SAg, SAs, SAp, PAg, PAs, and PAp.

Third, what decision structure should we use given the entries in the pattern tables? We examined several state machines, finally deciding on the 2-bit counter scheme of Jim Smith's 1981 ISCA paper.

Finally, how many bits of history should be kept in the history registers? Using n bits of history meant a pattern table of size 2^n. Adding one bit of history meant doubling the size of each pattern table.

We introduced the Two-Level Adaptive Branch Predictor at MICRO in 1991,[1] demonstrating that a lot more performance could be obtained with a branch predictor that was significantly more accurate than the previous best case. Our predictor showed that maintaining two levels of branch history would provide much greater accuracy than was possible with the simpler branch prediction schemes. Other researchers soon agreed that the benefit from branch prediction could be substantial, and that maintaining two levels of branch history was the correct mechanism. The result was a wave of papers on dynamic branch prediction, successively improving on our basic two-level predictor design. We followed our first paper with a more comprehensive paper describing alternative implementations of the two-level scheme in ISCA 1992,[4] and Kimming So and colleagues at IBM published their correlation predictor in ASPLOS later that same year.[5] Soon after, Scott McFarling introduced gshare,[6] a variation on the GAs two-level predictor and hybrid branch prediction.

At our industrial affiliates meeting the next year, Tse-Yu presented our branch predictor. Intel engineers in the audience were excited about its possibilities. They had decided to increase the pipeline of their next processor Pentium Pro from 5 stages to 13 to cut down the cycle time to be more competitive with all the emerging RISC designs. More pipeline stages meant a greater branch-misprediction penalty, so Intel badly needed a better branch predictor. Intel's interest sparked substantial interaction between us, resulting in Tse-Yu spending the summer of 1992 at Intel in Hillsboro, Oregon, where the Pentium Pro was being designed. Intel decided to opt for the PAs version of the two-level branch predictor and did all the hard engineering to make it implementable in the Pentium Pro. With the announcement of that product, the computer architecture landscape had changed forever in favor of aggressive branch prediction.

Tse-Yu Yeh completed his PhD at Michigan in 1993 and took a job with Intel in Santa Clara. Afterward, he joined Sibyte, which was acquired by Broadcom; later he moved to PA Semi and eventually to Apple. He has been responsible for many microprocessor designs over the years. He is currently an engineering director at Apple, where he is responsible for CPU verification. Yale Patt retired from Michigan in 1999 to become the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin. He continues to thrive on directing PhD students in microarchitecture research and teaching both freshmen and graduate students.

References

1. T.-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61.
2. M. Butler et al., "Single Instruction Stream Parallelism Is Greater Than Two," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, 1991, pp. 276–286.
3. J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Ann. Symp. Computer Architecture*, 1981, pp. 135–148.
4. T.-Y. Yeh and Y.N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, 1992, pp. 124–134.
5. S.-T. Pan, K. So, and J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 76–84.
6. S. McFarling, "Combining Branch Predictors," tech. note TN-36, Western Research Library, 1993.

Tse-Yu Yeh is an engineering director at Apple. Contact him at tyyeh@apple.com.

Yale N. Patt is the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin. Contact him at pattyn@austin.utexas.edu.

Retrospective on Code Compression and a Fresh Approach to Embedded Systems

ANDY WOLFE
Santa Clara University

In late fall of 1991, I was a first-year assistant professor at Princeton University. I was teaching an upper-level course on computer architecture based on quantitative analysis, a relatively new approach that grew out of research at Stanford and Berkeley in the prior decade. I was, quite literally, teaching the class "by the book" using the recently released text *Computer Architecture: A Quantitative Approach*.[1] I had been able to sit in on John Hennessy's similar course a few years earlier where he used the draft manuscript for the text, and although I was unable to replicate his gravitas, I was able to convey the primary message to my students: Computer architecture research had changed, and it was incumbent to be able to model, measure, and evaluate everything we did. Alex Chanin was a student in that class, on leave from AT&T in order to get a master's degree. He approached me mid-semester to discuss ideas for a thesis topic related to computer architecture. He made the primary requirements for a topic very clear. It had to be important enough that it would be accepted for graduation, but equally as important, it had to be finished on time at the end of the next semester so he could graduate, get married, and go back to his job on time. Working within those constraints, we developed the technology discussed in the paper, "Executing Compressed Programs on an Embedded RISC Architecture."[2]

I had a background in embedded computing, primarily using microcontrollers to design peripherals such as touchpads and touchscreens for PCs. I had decided to focus on this area when I joined Princeton, working with Marilyn Wolf and Sharad Malik on hardware/software codesign and on the emerging opportunities in digital audio and video. Alex had worked on telecommunications-based computing at AT&T. He repeatedly would remind me that the questions I was posing in class were not relevant to "real engineers with real jobs." A change in cache configuration that would improve average performance by a couple percent was not important where he worked. Engineering decisions were dominated by hard and soft limits on space, power requirements, cooling capacity, and cost. We started looking at the current state of embedded systems to find an interesting problem to solve. Could we improve cost, size, or power as compared to conventional processors? Could we show that we had made such an improvement with quantitative measures? And, of course, could we find a problem that could be "solved" in the next six months using the tools and resources we had on hand? With those goals in mind, we decided to address the question of whether we could make programs for reduced-instruction-set computing (RISC) processors smaller so that the cost of program memory could be reduced.

The system we developed was called the compressed code RISC processor (CCRP). The basic idea was that programs were compressed as the last step in program development using a block-based compression code. The compressed blocks were stored in the program ROM, using substantially less space than the original programs. In practice, this meant that we could fit more features and functionality into a limited-size program memory. At runtime, the compressed blocks are decompressed when read into the instruction cache. Neither the processor core nor the instruction cache itself need be aware that the programs were stored in compressed form. A line address table with its own cache provided pointers for locating the compressed blocks. Although the decompression time could add to the cache-line access time, in many configurations the reduced number of program memory accesses required to fetch the compressed data would compensate for the decompression time. In any case, decompression was required only for a cache miss, so any performance impact would be small.

In looking back at this research, I find it most interesting to look at the factors that led us to think about this problem and about the proposed solution. It is interesting to review not only what we published, but the other alternatives we rejected, because at least one of those has become successful. Most of all, however, I now find the specific problem that we addressed to be less interesting than the constraints that we put on the problem. On the basis of our limited resources and our desire to make the solution as palatable to industry as possible, we focused on using mainstream RISC CPU architectures in cost-sensitive embedded

systems. This was a different way of thinking about embedded CPU research at the time, but it reflected some important changes in the CPU industry and the way many researchers were starting to think about embedded computing. This may have been the most important impact of our paper, providing a concrete example of success to researchers who were starting to think about embedded computing in a new way.

The State of the Art

In today's world it seems obvious that a RISC-based system on chip (SoC) is a good solution for an embedded system, but when we started working on this research, such devices basically did not exist. Moreover, the trends in RISC processor development and mainstream CPU core development in general were moving further away from suitability in embedded systems.

As an academic and a computer architect, I had expected RISC and complex-instruction-set computing (CISC) implementations to begin to converge. During my time as a graduate student at Carnegie Mellon University, Bob Colwell had authored a well-known paper comparing fundamental aspects of RISC and CISC architecture.[3] I was highly persuaded by its primary conclusion, that at the most fundamental level RISC and CISC differ only in the way they represent and decode instructions. All other tenets of RISC architecture, such as large, general-purpose register files, pipelining, on-chip caches, and reliance on compiler optimization, were general advances in processor implementation tied to advances in VLSI technology. They were equally applicable to RISC and CISC architectures. By 1991, we were starting to see this convergence at the high end. Intel processors like the i486 and the upcoming Pentium had RISC-like microarchitectures and could match RISC processors in integer performance; however, in terms of the embedded processor landscape, RISC and CISC implementations were starting to diverge in disturbing ways.

Embedded systems developers focused on low cost, low power, high levels of integration, and absolute predictability at runtime. The heart of the embedded systems market was the 8-bit microcontroller. These chips, such as Intel's 8051 and Motorola's 68HC11 and their derivatives, included on-board program and data memory, integrated analog and digital I/O peripherals, on-chip clock generators, serial ports, timers, interrupt controllers, and low-power sleep modes. They typically used a fraction of a watt at peak performance and were provided in low-cost plastic packages with 40 to 80 pins. The processors were cheap. Systems built with them were cheap. They were small and required little power. Moreover, they were highly predictable. Developers would "cycle count" the critical paths in programs to determine performance, because the execution time of each instruction in the path was known and fixed. This allowed them to be used for real-time control. At the higher end of the embedded market, there were newer 16-bit microcontrollers that resembled the 8-bit parts at a higher cost and simple single-board computers based on older 8086 or 68000 family processors. These were smaller than PCs and lower power but also very predictable in operation. Although these solutions were suitable for many applications, it was clear that there was interest in higher performance and more capabilities. General-purpose computers were doubling in performance every year, and the embedded world was being left behind.

Almost all of the activity in CPU development focused on high-performance RISC processors for workstations and servers and RISC-like microarchitectures for CISC chips that could compete with them. In the past, CPU implementations would often start off as desktop processors, and then, as they became cheaper, they would be used for high-end embedded systems and then reused in integrated microcontrollers. This pathway seemed to be at an end. Performance was skyrocketing, but the very features that were contributing to these performance increases made them increasingly unsuitable for embedded use. The most prominent of these features was the heavy dependence on on-chip caches to support high clock speeds and the high instruction bandwidth needed to provide one or more 32-bit instructions per clock. The problem for the embedded world was that nobody understood how to use a processor with caches in a real-time environment. We thought of caches as having probabilistic performance, and anyone could imagine a worst-case scenario in which a processor with caches might be 10 or more times slower than the typical case. This led to lots of ranting and raving in the embedded-systems community about missiles that wouldn't fire, car engines that would explode, and airplanes that would drop out of the sky. It's hard to believe in retrospect, but many smart people in 1991 thought that real-time embedded systems simply could not use processors with caches, and that there would be very limited further improvement in embedded processor performance.

Additionally, the caches that could fit on a chip at the time were never large enough to keep up with the processor core's performance, and thus designers would try to fill up as much of the chip as possible with cache. This meant no room to integrate any other features, such as I/O, main memory, clock generators, or communications. Unlike the early days of the microprocessor, when Intel, Zilog, and Motorola would create an ecosystem of peripheral chips to support each new processor bus, these new RISC microprocessors were targeted toward large computer manufacturers that would develop proprietary support chips for each computer system. Building a complete system using the new RISC microprocessors required a much larger investment. Other factors made the cost of these microprocessors unsuitable for embedded use. As RISC architectures, they relied on high clock speeds and lots of instruction and data bandwidth. Although caches helped, these

processors were in large ceramic packages with lots of I/O pins and power and ground pins. The SuperSPARC processor had about 300 pins. The R4000SC had about 450 pins. Even the low-end RISC processors had close to 200 pins. This meant that these were expensive parts that used lots of I/O power for off-chip communications. Designers also focused on clock rate at the expense of power. Many of these RISC CPUs used dynamic logic and thus (at the time) did not have a low-speed or sleep mode. Many designs (such as Pentium and SuperSPARC) had moved to Bi-CMOS circuits. Unlike today, where power is a major constraint, power had become a tool for increasing clock speed.

Along with all of these CPU implementation issues, embedded system designers were deterred by one fundamental aspect of RISC architectures—that they used fixed-length, 32-bit instructions. This, along with the fact that RISC compilers were all tuned for performance, meant that in practice RISC object code was much larger than CISC code. Some measurements showed that, for example, Sparc code could be three times as large as VAX code for the same program. This difference was not as pronounced for other CISC microprocessors, but there still seemed to be about a 50 percent code-storage penalty for using a RISC architecture as compared to CISC. In a system with a limited dollar or space budget for program ROM, this meant that a RISC system could include only two-thirds as much functionality as a CISC system. This seemed like a major disadvantage. Remember, at the time, a typical midrange embedded system might only have 32 Kbytes of program memory.

Given all these factors, as well as the fact that most RISC products were being promoted as high-end solutions, designers did not seriously consider RISC to be practical for embedded computing. Some earlier efforts had attempted to introduce embedded RISC architectures, most notably the AMD 29000 and the Intel i960, but these were low-integration, high-pin-count devices, mostly targeted for laser printers. Both teams were known to be working on more expensive superscalar versions of these chips rather than bringing them downmarket. In fact, as late as 1992, Intel still included the i960 in its Multimedia and Supercomputing databook, along with the i860 and i750, as opposed to in its two-volume Embedded Microcontrollers and Processors databook. ARM was deep into the development of the ARM610 for the Apple Newton, but the idea of an embedded ARM core as part of a third-party SoC was not well-known until several years later with the introduction of the ARM 7 core.

The embedded systems community was stuck in a rut. It was clear that the semiconductor industry was investing all of its money and effort into RISC and yet it was not at all clear how embedded systems would ever benefit from this investment. This was the problem we were trying to solve when we developed the CCRP.

Developing an Approach

By 1991, I had been struggling with this problem for some time. I had been looking for some unique architectural approach that would break past existing performance constraints. One thought was that if we brought the instruction set and datapath closer to the application level and replaced runtime interpretation with compile-time optimization, we could reduce code storage space and bandwidth and devote more hardware to calculation. As one of those projects, in 1988 I developed a computer based on the new Xilinx field-programmable gate array (FPGA) technology based on reconfigurable instruction decoding and datapaths. Early results were promising. Application-specific instruction sets could be very dense, and increases in instruction-level parallelism (ILP) of two to four times over conventional processors were easy to develop. I packaged my initial results into a paper to be presented at MICRO 21.

For the 1988–1989 academic year, I was invited to be a visiting graduate student in Ed McCluskey's lab at Stanford. I presented the MICRO 21 paper to some of the Stanford faculty and students in the Computer Systems Lab. After that presentation, Mark Horowitz commented that an architectural improvement of two to four times was immaterial compared to a modern RISC processor. He explained that general-purpose RISC processors benefitted from enormous engineering efforts, including critical-path circuit tuning and access to the newest manufacturing processes. He claimed that this design effort, which was only affordable in a high-volume, general-purpose processor, would provide 10 to 100 times improvement in cycle time compared to my FPGA approach and that alternative application-specific approaches made sense only if they could exceed that. I basically disregarded the comments at the time as simply a defense of the status quo, but as I continued my research over the next couple of years, it became clear that he had observed a fundamental change in the computer industry. Because processors were becoming exponentially more complex, the ability to invest large amounts of money in their development was becoming as important as the core architectural ideas.

During the summer of 1989, my advisor, John Shen, arranged an internship at ESL/TRW to design a proof-of-concept system to track incoming missiles as part of the Strategic Defense Initiative. I started working on a VLIW-style board-level processor based on high-performance floating-point units (FPUs) from Weitek and microcoded instructions. Our objective was to get maximum performance within a dorm-fridge-sized 9U cabinet with a development budget that I think was a few million dollars, including application software. Sometime midsummer, I read an *IEEE Micro* article describing the upcoming Intel i860XR 64-bit microprocessor[4] and then quickly obtained the databook and programmer's manual from Intel. The i860XR was expected to ship by September with the ability to do 80 Mflops, double-precision,

at 40 MHz with a promise of a 50 MHz part soon behind. Dr. Horowitz's prediction had already come true. The floating-point engines we could buy topped out at 20 MHz and did not have enough pin bandwidth to sustain double-precision calculations at that rate. If we built our own chips, we estimated that we could sustain 20 MHz, but even with substantial ILP, it would be difficult to reach 80 Mflops, and chip development would cost millions of dollars. Intel simply had been able to outinvest anything we could duplicate. They used a bleeding-edge 1,000-nm CMOS process (no giggling from millennials) and tuned the circuits using the best designers. They customized the memory technology in the caches, included DRAM support, and developed compilers tuned to the architecture. They raised the bar on everyone's expectations, and did it in a processor that used 3 W. We did not know how to duplicate the performance per unit area, but at the same time we did not know how to build a reliable real-time system using a processor with such complex pipelining and heavy reliance on caches. I decided to prove to my team that I could "tame" a modern RISC microprocessor by using restrictive Fortran-style programming, modeling the cache misses and DRAM page misses, and limiting interrupts. We ended up building a four-processor-per-card message-passing architecture that provided up to 320 MFlops per board. I had "tamed" a leading-edge workstation processor and used it as a practical digital signal processor in an embedded system. Were there other ways that this approach could be used to leverage the investments in high-performance RISC processors while maintaining the characteristics of embedded systems?

Developing a Solution

When Alex and I started to look for an embedded system problem to solve, I thought back to the Colwell paper about RISC and CISC. One of the problems in using RISC processors in embedded systems was that the code was so much bigger. Even in a higher-end system like a laser printer or an advanced car engine controller, which might benefit from the added performance, there often was no room and no budget to add 50 percent more ROM chips. The RISC code was simply too big; but the question was why? At a fundamental level, RISC code and CISC code were specifying the same amount of information. They were describing the same amount of computing to be done. There should be some way to express this in the same number of bits. If we were willing to accept a performance impact with RISC, could we make the code as small as CISC code? How big would that performance impact be?

We started to explore various ideas. One idea was to write CISC code for an existing architecture, then decode it into RISC code on instruction fetch. This was inspired in part by the decoded instruction cache that Dave Ditzel and Alan Berenbaum had developed in CRISP.[5] We could use an existing CISC compiler and retain most of the RISC core—everything after the decode stage. However, we quickly lost enthusiasm when we looked at some details. There would be no way to specify and efficiently use all of the registers in the RISC. Also, at the time, we believed that much of the magic in RISC architecture was the ability to compile for one specific pipeline. We would lose that ability. We moved on to other ideas.

I was enamored with the idea of using a 16-bit instruction set that used a limited set of processor resources, used only the most common opcodes, supported only short immediates, and specified two registers instead of three. A mode-switch opcode would switch between short-instruction and long-instruction mode. I thought that we could get close to a 2:1 improvement in code density. Alex prototyped the system, but we couldn't get it to work to our satisfaction. Performance dropped 30 to 50 percent, and the extra instructions meant that the code was not much smaller. I suspected that the real problem was that we were just not doing a good job of rewriting the program with the new instructions, and that a new compiler was needed. I had lots of ideas I wanted to try, but none would fit into Alex's "graduate this semester" timetable, so we put it on hold and decided to try our remaining approach. Within two days, Alex was already getting promising results. Of course, ARM successfully implemented an alternative 16-bit instruction set with their Thumb instruction set that was announced around 1995.

The final approach was to just compress the program and then decompress it at runtime. We ran some programs through "compress" on a Unix system and got 40 to 50 percent size improvements. Of course, one problem we faced in executing compressed code is that we no longer knew where the code for any branch target had ended up. If we tried to patch branch targets in the compressed code with the new locations, the compression would change and the locations would move.

Our first inclination was that we would decompress the programs into RAM and run them uncompressed, but upon reflection that made little sense. The extra RAM required to hold the program would offset any savings in ROM. At that point, we came back to the idea of the decoded instruction cache. If the decompression happened on cache refill, then the cache could be our decompressed program RAM. Every RISC core would have an instruction cache. Since the cache would hold only uncompressed code, the CPU core would not even need to know about the compression and would need no modification. In fact, the cache itself would need no modification other than the refill engine, which would decompress the instructions. This was a great idea, but it quickly led to the next problem. Active cache lines were dispersed throughout the program. You could not decompress a "compress" file (which used the Lempel-Ziv-Welch algorithm) unless you started from the beginning and proceeded sequentially. The solution was to

select a moderately large cache line size and to compress each correspondingly sized program block separately. That way we could decompress a single cache line at a time. For such small datasets, a preselected Huffman code worked well. We looked at various alternatives but decided that for the MIPS R2000 code in our experiments, treating each byte as a character (rather than each instruction) worked best.

The remaining problem was how to find a cache replacement line in the compressed code. Each block was no longer at its original address in memory because of the compression. We decided to incorporate a table of pointers to each cache line, similar to a page table that locates pages on a disk. The table would be stored in ROM with the code and could be referenced to find any block of the program. We included a cache for the table, a cache line address look-aside buffer (like a translation look-aside buffer for a page table), that would eliminate the extra memory read. Even with all the overhead of block coding and the tables, we could still substantially reduce the code size.

We had all the pieces for a workable system. We built a simulation, measured compression rates, and simulated performance. We wrote up the results into a paper, and Alex prepared his master's thesis. All we needed were the final two keys to publishing a computer architecture paper in 1992: we needed to get RISC into the title, and we needed a four-letter acronym for our project.

Aftermath

The paper was well-received and inspired many researchers to improve on our methods. There were more than 100 follow-on papers that improved coding efficiency and system performance. At least two processor design teams implemented the technique into commercial processors. Josh Fisher and Paolo Faraboschi included code compression in their LX VLIW processor at Hewlett Packard. IBM incorporated it into the CodePack technology in certain PowerPC processors. However, in looking back at the effort and its impact, what I find most interesting is that it represented a change in thinking about how to do embedded systems architecture research. Many researchers with expertise in high-performance architecture began to think about how the investment in high-performance processors could be leveraged into the embedded space. Researchers who saw this work presented at MICRO and other presentations, as well as other researchers who simply noted the same technology shifts through their own experiences, began to propose other problems that could be solved using a similar approach. This reinvigorated embedded systems as a research topic and led to widespread advancement in the following years.

In retrospect, this change in how we thought about embedded computing was inevitable. The changes that I observed were rapidly impacting the entire industry as the cost and complexity of developing processors and compilers escalated. The development of the ARM7 and eventually the synthesizable ARM7 TDMI had likely started. This would lead to the development of other licensable peripheral cores and buses that would be used with ARM cores to build RISC microcontrollers and SoCs for embedded systems. Kurt Keutzer at Berkeley was also already working on code compression techniques for RISC. Other research groups began exploring other ways to leverage mainstream RISC architectures for embedded systems applications and thus developing system-wide approaches to managing power and predictability. The availability of open source tools such as compilers, simulators, and operating systems increased the opportunities for embedded systems research in both academia and industry. Interestingly enough, we are now reaching a point where the development model is beginning to switch. The embedded market—including general-purpose devices such as mobile phones that have severe cost, power, and packaging constraints and real-time requirements—now dominates the processor market. Embedded designs are evolving into servers and infrastructure in which cost, power, and packaging are important. Now we need to learn how to leverage the investment in embedded computing to support the general-purpose computing markets. MICRO

References

1. J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1991.
2. A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, 1992, pp. 81–91.
3. R.P. Colwell et al., "Instruction Sets and Beyond: Computers, Complexity, and Controversy," *Computer*, vol. 18, no. 9, 1985, pp. 8–19.
4. L. Kohn and N. Margulis, "Introducing the Intel i860 64-Bit Microprocessor," *IEEE Micro*, vol. 9, no. 4, 1989, pp. 15–30.
5. D.R. Ditzel, H.R. McLellan, and A.D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, 1987, pp. 309–319.

Andy Wolfe is a consultant and an adjunct professor at Santa Clara University. Contact him at awolfe@awolfe.org.