The 2014 MICRO Test of Time Award Winners: From 1978 to 1992

Onur Mutlu
Carnegie Mellon University

Rich Belgard

As you may know, the International Symposium on Microarchitecture (MICRO)—the flagship microarchitecture conference, and a premier computer architecture conference for nearly five decades—selected 10 papers as recipients of the first set of MICRO Test of Time (ToT) Awards in December 2014. We announced the winning papers and described the selection process in the March/April 2015 issue of *IEEE Micro*.[1] The authors of these 10 distinguished papers were invited to write short retrospectives to reflect on their work, which was done at least 20 years ago. This issue features retrospectives written by the original coauthors of two of the award-winning papers. We briefly introduce these papers and retrospectives, and we hope that you will enjoy reading them as much as we have.

The first retrospective is for the oldest paper that won the 2014 MICRO ToT Award. "Microprogrammed Implementation of a Single Chip Microprocessor" by Skip Stritter and Nick Tredennick was published in MICRO 1978.[2] It introduced the idea of a two-level control store (textbook material in computer architecture today) with the goal of minimizing the chip real estate dedicated to the control logic used in microprogrammed processor designs, and in particular the memory used to store the microinstructions. The two-level control store is essentially two carefully codesigned programmable logic arrays (PLAs) that together comprise more compact storage for microinstructions than a single monolithic control store. It was born from the necessity to "maximize the contribution of every transistor spent" (to quote Tredennick's retrospective) in the design of the Motorola MC68000 processor. The paper also described in detail the microprogrammed control logic implementation of a single-chip microarchitecture, based on the MC68000 experience. In his retrospective, Tredennick describes his experience at Motorola that led to this paper and discusses his subsequent experiences in industry, which were partially shaped by his involvement with the MC68000. He also muses about the connection between design and design automation processes, which makes the retrospective a fun historical perspective and a delightful read for the *IEEE Micro* audience.

The second retrospective is for one of the youngest papers that won the 2014 MICRO ToT Award. "Code Generation Schema for Modulo Scheduled Loops," authored by Bob Ramakrishna Rau, Michael S. Schlansker, and P.P. Tirumalai, was published in MICRO 1992.[3] The paper provides a "recipe book" (to quote Schlansker) that discusses and enumerates code-generation choices for correctly and efficiently optimizing instruction schedules of loops for various architectures, including very long instruction word (VLIW) and superscalar. The paper covers architectures incorporating a varying set of features for loop schedule optimization, using the notions of software pipelining and modulo scheduling. The work is based on the authors' extensive (about a decade long) experience in hardware/software codesign for realizing Cydrome's Cydra 5 processor. Yet, the paper's loop code scheduling strategies apply far beyond the extensive architectural support provided by the Cydra 5 for loop scheduling purposes, as Schlansker's retrospective beautifully describes.

As we conclude, we would like to take the opportunity to pay tribute to the extremely valuable impact that Bob Rau has had in our field, especially in the development of compiler technology and VLIW processors, as well as hardware/software cooperation in instruction-level parallelism. It has been 13 years since Bob died, but his impact is wonderfully felt in the compiler technology commonly in use today, along with the many clearly articulated technical articles he contributed to academic literature. His works are taught in many modern compiler and computer architecture

Published by the IEEE Computer Society

0272-1732/16/$33.00 © 2016 IEEE

classes today. Bob was one of the most prominent contributors to MICRO for decades, and our selection of the 1992 article, for which he was the primary driver (according to Schlansker's retrospective), as part of the first set of MICRO ToT Awards points to the technical excellence and value of insight he upheld as a leading member of our community. We hope these two key values continue to thrive as microarchitecture/architecture and hardware/software codesign become even more important with fundamental challenges threatening the large improvements obtained from the scaling of the underlying circuit and device technologies. MICRO

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. S. Stritter and N. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," *Proc. 11th Ann. Workshop Microprogramming*, 1978, pp. 8–16.
3. B. Ramakrishna Rau, Michael S. Schlansker, and P.P. Tirumalai, "Code Generation Schema for Modulo Scheduled Loops," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, 1992, pp. 158–169.

Onur Mutlu is the Strecker Early Career Professor at Carnegie Mellon University. Contact him at omutlu@gmail.com.

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.

Evolution of Microprocessor Logic Design

Nick Tredennick
Jonetix

In the summer of 1977, I was teaching as an assistant professor at the University of Texas in Austin when Tom Gunter walked into my office and introduced himself. He asked if I'd like to work for Motorola on a microprocessor design project. My areas of expertise were computers and logic design, and I had a little experience with microprocessor applications, but no experience with microprocessor design or semiconductor design. Nevertheless, there was mutual interest and I took a job with Motorola beginning in September 1977. The project was a next-generation microprocessor design called MACS (Motorola Advanced Computer System). Motorola's previous microprocessor designs had been 8-bit accumulator-based designs suitable for embedded applications; MACS was to be a 16-/32-bit design more suitable for computer applications. Tom said he eventually wanted me to work on the design of the on-chip cache, but that he first needed me to begin work on the microprocessor's logic design "until we find a competent logic designer." Of course, that never happened, and I spent my time doing the logic design for what became the MC68000.

I began looking for books and articles on microprocessor logic design. I was unable to find documentation for any microprocessor logic design methods. That seemed odd, given that the Design Automation Conference was already 14 years old in 1977. Just what processes were all those software engineers automating? Well, OK, I'd have to make up the design process as I went along.

At the time, the biggest differences between computer design and microprocessor design were in the constraints placed on the microprocessor's designer. The microprocessor's entire design had to fit on a single power-, pin-, and transistor-constrained, size-limited silicon chip.

All of the microprocessor's computing resources (data registers, address registers, program counter, and arithmetic units), interrupt logic, interface logic (pin and external bus control), and control logic had to fit inside the transistor, area, and power budget. Since we began by doubling or more than doubling the width of the data and address registers and the arithmetic units, as well as substantially increasing the number of data and address registers compared to an 8-bit accumulator-based design, we quickly ate into the transistor-budget increases provided by our move to the next most advanced semiconductor process. The consequence of these decisions was that, in the implementation of the control logic, we had to

maximize the contribution of every transistor spent.

In 1977, moving to the next semiconductor process meant designers had somewhat more than twice the number of transistors enabled by the previous-generation semiconductor process. With each semiconductor process generation—which came along about every 18 months—transistor area shrunk by half, which doubled the number of available transistors in a fixed area. Additional available transistors came from improvements in transistor layout and from lithography advances that enabled production of larger chips.

As transistor size decreased, power per transistor fell, so that chip power rose only slowly. Leakage currents, even for new process generations, were negligible, so we worried only about active power. In addition, the smaller transistors were faster, so clock speeds rose and performance increased with each semiconductor process generation. The 18-month cycle for new semiconductor process generations also drove the development cycle. Project delay could mean that your competitors benefitted from twice as many of the newer, faster transistors in their designs.

Design process

The MC68000 was probably among the last of the pencil-and-paper microprocessor designs. The project did not have the benefit of either computer-aided design entry or computer-based logic simulation. I drew pencil-and-paper diagrams of the execution units, decoders, logic units, and interconnections. I used modified Karnaugh maps (of up to 16 variables) for logic minimization. I wrote register transfer sequences in cycle-by-cycle flowcharts for each instruction in pencil on large sheets of paper.

I used these methods both to place and to assign the instructions' op codes (for efficient instruction decoding and uniform access to register fields) and to optimize the programmable logic arrays (PLAs) that decoded the instructions and provided instruction execution sequencing.

For control efficiency, instructions shared operand address calculation sequences. One instruction decoder pointed to the operand address calculation sequence and a second pointed to the required operation sequence. The address calculation sequence computed the operand address and sent a request to memory for the operand before transferring control to the second instruction decoder, which performed the operation and stored the result. This led to a problem with the clear memory instruction, which read the location to be cleared before writing a zero to that location. It was necessary to enable universal sharing of the address calculation functions, but some users didn't expect a read to accompany a clear memory instruction.

Similar to the sharing of address calculation sequences, instruction operations shared sequences. Add, subtract, AND, OR, and XOR, for example, could all share a common two-operand arithmetic sequence through the use of an arithmetic logic unit and condition-code control table. For that table, the instruction decoder selected a row and the common two-operand arithmetic sequence chose a column; that way, the sequences could be common and the operations different.

The controller for the MC68000 microprocessor looked like a two-level control store with vertical (compact) microcode for sequencing and horizontal (mostly decoded) microcode for execution unit control points. The structure is described in the paper "Microprogrammed Implementation of a Single Chip Microprocessor," which Skip Stritter and I wrote. What we called vertical and horizontal microcode are nothing more than the outputs of two highly optimized PLAs operating in parallel. The execution unit control PLA was optimized to eliminate duplicate states that would have occurred if the control points had been included in the sequencing PLA.

Most of the decisions in the design of the control unit focused on transistor efficiency.

Logical conclusions

At the end of the MC68000 project, in late 1979, I resolved to do two things. First, I decided to make a list of the problematic design decisions that I had made during the project, so that I could avoid those errors in the next design. Second, because available design tools did not support the design method I had been using, I decided to create a design description that could act as the basis for implementing design aids.

Make better decisions

At the end of the MC68000 project, I made a list of the design decisions that I felt led either to inefficiencies that could have been avoided or to increased difficulty in completing the design. I don't recall the contents of the list, but I believe there were about 10 items. The character of the items on the list was something like "instead of X as a method for register decoding and control, Y is probably to be preferred." I resolved to use this error-correction sheet at the beginning of my next microprocessor design.

About a year later, I made use of that list when I began the Micro/370 microprocessor design while working at IBM Research in Yorktown Heights. And here's why I don't recall the contents of that list: at the end of the Micro/370 design project, when I made a list of the design decisions that I felt led either to inefficiencies that could have been avoided or to increased difficulty in completing the design, it turned out to be essentially the inverse of the list that I had made at the end of the previous design. My lesson from this was that it's probably not fruitful to try to judge your design decisions in retrospect, because it is impossible to forecast the consequences of the alternatives in the absence of actually doing the detailed design work to implement them.

Design process, design automation

In 1979, design automation was a popular and growing business, but there seemed to be little correlation between what was being automated and the actual logic design process that I had been using. It looked more like design engineers were

modifying their design processes to conform to the available tools. That seemed like an inefficient approach to me, so I resolved to document the microprocessor design process that I had used so that if there was any interest in automating an actual design process, there would be at least one template for doing so. About a year after I went to work for IBM, I began documenting the process I would use to design a microprocessor. Since I couldn't use the information from the MC68000 design—because many of the design details were confidential—I began a new IBM 360-based microprocessor design that was eventually named Micro/370.

It began as an example design, but it grew into a team building a real microprocessor when we decided we had to actually build the microprocessor for the design process to have credibility. The project produced a Micro/370 microprocessor that was functional and even booted IBM's VM operating system, but was not successful in the market. The project also resulted in *Microprocessor Logic Design* (Digital Press, 1987), a computer engineering textbook that described the design process. A number of universities adopted the text for courses.

Constraint evolution

So, there we were in 1987—10 years after I had gone to work at Motorola and had been unable to find a microprocessor design process, there now existed a documented microprocessor design process. Would the design automation engineers finally see the light and begin automating a process from a design engineer's template?

In a word: no.

It didn't happen and it shouldn't have happened. I was disappointed, but I shouldn't have been. Just as I had done at the end of the MC68000 project's "design errors" list, I was taking the same myopic view of the design process. We all took for granted that transistors shrunk and got faster with each generation. My mistake was that I took as an unstated assumption that the design process didn't evolve with the march of semiconductor progress. That was unwise. The design process changed dramatically as transistors shrunk because the constraints changed.

The design process I used was suited to a single individual controlling an entire logic design encompassing fewer than 100,000 transistors. In computer architecture terms, the processors of 1977 were primitive even compared to the mainframe designs of the time. Microprocessors didn't have to invent architectural features; we were still copying features from the more advanced mainframe implementations of the time.

In the time it took me to initiate and complete another design and write the process, microprocessor designs were already incorporating millions of transistors. Computational path widths would at most double and then level off, so execution units became a smaller portion of the design; there were plenty of transistors to spend on control logic, easing constraints on the efficiency of controller design. On-chip caches, floating-point capabilities, and multiprocessor designs debuted to consume excess transistors.

Microprocessor development accelerated past mainframe and minicomputer design as the leading edge of computer architecture, encouraging innovators to enter the field. Newer design projects lead to specialization in design expertise in areas such as cache replacement strategies, branch prediction, and floating-point implementation. This further specialization led to the growth and fragmentation of design teams, which called for more coordination and for standardization of methods and design documents.

Progress in semiconductor process conspired to change design constraints. As transistors shrunk below 90 nm, leakage currents became significant, changing the design emphasis in power management. Logic speed diverged further from memory speed as semiconductor process engineers emphasized speed for logic and density for memory, changing constraints in cache design and shifting emphasis to consistency issues in the cache hierarchy. Relative differences in logic speed and propagation delay forced tradeoffs in pipeline depth versus per-stage logic processing and in on-chip location of functional units. Multiprocessor designs opened whole new vistas of requirements for innovation and development.

I thought there should be a core method for microprocessor logic design and that should be the basis for the automation of design aids. In an environment evolving rapidly with progress in semiconductor process, that assumption was invalid. I built an ad-hoc design method that was suited to the constraints present at the time I began the design. Maximum chip size could accommodate fewer than 100,000 transistors, making transistors the scarce resource, so design efficiency was paramount. Today's chips easily contain billions of transistors; transistors are abundant, so the scarce resource is designers or design management, verification, or time. Newer microprocessor design projects require large teams and emphasize specialization and design fragmentation.

In the 1970s, microprocessor design was primitive. Its logic design, incorporating a few tens of thousands of transistors, could be managed by a single person or a small team, and in terms of computer architecture, it was a trailing-edge implementation incorporating features that had been pioneered in mainframes, minicomputers, and workstations. In contrast, today's microprocessor design is perhaps the most sophisticated of all engineering design. Its logic design, incorporating billions of transistors, requires a large team of experts in a wide range of specialties, and in terms of computer architecture, these advanced microprocessors are blazing the trail in innovation. MICRO

Nick Tredennick is a VP and engineer at Jonetix. He is a life fellow of IEEE. Contact him at bozo@computer.org.

Efficient Code Generation Schema for Modulo Loops

MICHAEL SCHLANSKER
Hewlett Packard Enterprise Labs

This retrospective is dedicated to Bob Rau, who sadly cannot participate in receiving this honor. It was Bob's passionate interest in computer architecture, instruction-level parallelism, and compilers that enabled our original paper, "Code Generation Schema for Modulo Scheduled Loops," with coauthor Partha Tirumalai.[1] This paper was published in 1992 at MICRO 25. It addressed a basic question—how can we efficiently execute loops on a processor with instruction-level parallelism while preserving processor simplicity? The paper condensed a body of work that resulted from Bob's passionate effort over an extended period of time into a summary of known techniques for loop acceleration. It distilled contributions by Bob, his close collaborators, and a broader parallel processing community into an architecture for, and a manual of facts about, synchronous parallel loop execution. The goal of this retrospective is to share a few highlights in our progress toward developing this body of work.

In addition to Partha Tirumalai and myself, other collaborators who worked directly with Bob and influenced this work included Bob's ESL collaborator Chris Glaeser; Bob's Cydrome collaborators, including Joe Bratt, Peter Donovan, Peter Hsu, Ross Towle, and Art Sorkin; and Bob's Hewlett-Packard collaborators, including Vinod Kathail, Meng Lee, and Scott Mahlke.

The Code Generation Schema paper resulted from Bob's lasting interest in instruction-level parallelism. An important launching pad for our work came from earlier work on single assignment languages and dataflow, such as MIT's tagged-token dataflow architecture,[2] which formalized the renaming of variable instances within a sequence of loop iterations in order to allow concurrent processing. Dataflow strengthened our understanding of loop-level parallelism, recurrences, and loop-carried dependences and helped us identify a rigorous means to express loop parallelism within our compiler's intermediate form.

But dataflow was not the target of our research. We wanted to accelerate innermost loops with synchronous hardware to exploit large amounts of parallelism without the complexities of hardware-based dynamic scheduling. Much as in earlier horizontal microcode, or very long instruction words (VLIWs),[3] our goal was to remove complexity from the processor and instead use a compiler to produce highly optimized static code schedules that are executed by simple hardware. Bob's early work toward this goal resulted in the Polycyclic architecture,[4] which combined wide synchronous execution with innovative shift-register hardware. The architecture demonstrated cyclic code schedules, with a period called the *initiation interval* (II), which controlled the steady state execution pattern for a loop. The innermost loop was scheduled as a cyclic pipeline of overlapped loop iterations. Code running these loops came to be known as software pipelines, and the compile-time scheduler used to generate code for such loops was called a *modulo scheduler*. The highest performance was achieved by identifying a minimal loop II, which accommodated throughput limitations that were dictated either by fully utilized computational resources or by the latencies of operations around a cycle of carried dependence.

Because of Bob's interest in developing efficient hardware for a new processor product (the Axiom processor was later renamed as the Cydrome processor), he strongly desired to replace the Polycyclic's shift registers with more efficient static RAM. Cydrome's Cydra 5 processor incorporated rotating register hardware to implement register renaming. For each operation, register addresses could be dynamically computed by adding a register offset that was specified by the operation to an iteration control pointer (ICP). The ICP was incremented by the execution of a loop branch to relocate register references within innermost loops in a manner similar to laying out a new context frame in dataflow, or incrementing a vector register pointer. This allowed code from parallel loop iterations to access values stored in a nonshifting RAM without replicating loop code. For machines without register renaming, a compiler uses Modulo Variable Expansion, which combines compile-time register renaming with code replication to enable parallel loop execution.

In addition to rotating registers, the Cydra 5 provided predicates to support conditional execution. A compare operation computed a predicate that could conditionally nullify operations that were dependent on that predicate. Predicated execution supported the if-conversion of conditionals in the body of loops, which allowed the parallel execution of conditionals without code replication.

Another complex problem is that of controlling the software pipeline fill and drain process. As execution ramps up,

reaches a steady state, and then ramps down, a different set of operations are either needed or unused in the software processing pipeline. Again, the Cydra 5 solved this problem without code replication. After a compiler determined an II and generated a cyclic code schedule, operations within a loop could be separated (according to their scheduled time) into stages, each lasting II cycles. The total number of stages, or stage count, indicates the maximum number of loop iterations that are simultaneously in process when a software pipeline reaches fully busy execution. The Cydra 5's loop branch operation computes a sequence of predicate values that are applied to code in each stage. The actions of the branch operation, which computed a stage predicate and advanced the ICP, caused code in each stage to be correctly executed or nullified according the software pipeline's fill and drain progress. This was called "kernel-only code" and eliminated the loop unrolling needed in other approaches.

Another important high-performance loop feature is speculative execution, which allows operations to be executed in scheduled code before it is known whether they would have executed in the original sequential code. Speculative operations produce errors that should not be reported until it is known whether they execute in the original code; they should be reported after that time. Speculative execution was used by Multiflow[5] and was incorporated into the IMPACT architecture[6] at the University of Illinois. Speculative execution is particularly important in *while* loops, in which load operations must be moved above one or more conditional exit branches to achieve good performance.

This discussion sets the stage for the Code Generation Schema paper's primary goal of developing efficient code generation schema for loops that execute on a range of hardware architectures with some or all of the loop features described earlier. We knew that many processors would not have rotating registers or predicated execution and that code replication could be used in their absence. For a statically scheduled VLIW with none of these features, achieving the highest performance requires substantial code replication. For complex-instruction-set computers, reduced-instruction-set computers, and superscalar machines that do not have advanced loop features or cannot perform sufficient dynamic scheduling in hardware, their compilers can still improve performance by exploiting schema for code rescheduling, register renaming, and replication. However, the amount of code grows with the amount of parallelism, and a large expansion in code size can degrade instruction processing performance.

Each hardware architecture choice presents complex compile-time code-generation tradeoffs between the amount of code replication and the achieved performance as a function of the loop's trip count. For example, loop preconditioning could be used to sequentially execute a residual number of iterations modulo p and then retire parallel groups of exactly p unrolled loop iterations to complete loop execution. However, this approach leads to poor performance with small loop trip count. Our goal was to develop a set of precisely defined code generation choices for a variety of important hardware architectural alternatives and compare these loop control schema for a specific amount of instruction-level parallelism as expressed with example pipelined multifunction datapaths.

In summary, our goal for this paper was to provide a recipe book for optimizing the schedule for loop codes for various architectures. The paper uses advanced Cydra 5 loop control features that eliminate a need for code replication. It defines code generation schema that can be used for various VLIW and conventional processors. The paper measures key attributes such as achieved performance versus loop trip count and required code size for varying degrees of hardware parallelism. Finally, it shows that without adequate hardware support, compile-time loop scheduling causes significant code growth unless performance is sacrificed for short trip-count loops.

I thank those who recognized this work as a lasting contribution to a large body of exciting research in instruction-level parallelism. Again, I would like to express my sadness that Bob Rau cannot join us to celebrate this honor. Bob was the primary driver for a body of work that was developed over much of a decade and culminated in the Code Generation Schema publication. We miss his unwavering pursuit of technical excellence, which was directed toward developing next-generation computer architectures to exploit instruction-level parallelism. He cultivated a positive approach to technology development based on enthusiasm, creativity, deep intellectual discourse, and perseverance that many of his coworkers remember fondly. MICRO

References

1. B.R. Rau, M.S. Schlansker, and P.P. Tirumalai, "Code Generation Schema for Modulo Scheduled Loops," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, 1992, pp. 158–169.
2. Arvind and V. Kathail, "A Multiple Processor Data Flow Machine that Supports Generalized Procedures," *Proc. 8th Ann. Int'l Symp. Computer Architecture*, 1981, pp. 291–302.
3. J.A. Fischer, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, July 1981, pp. 478–490.
4. B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Ann. Workshop Microprogramming*, 1981, pp. 183–198.
5. R.P. Colwell et al., "A VLIW for a Trace Scheduling Compiler," *IEEE Trans. Computers*, vol. 37, no. 8, 1988, pp. 967–979.
6. S. Mahlke et al., "Sentinel Scheduling for VLIW and Superscalar Processors," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 238–247.

Michael Schlansker is a Distinguished Technologist at the Hewlett Packard Enterprise Labs. Contact him at mike_schlansker@hpe.com.