Preface

Lin Chao
Editor
Intel Technology Journal

This Q1 2001 issue of the Intel® Technology Journal covers the very old and the very new at Intel. The first paper reminisces about early chip development efforts at Intel. Have you ever wondered why Intel's microprocessors were named 8086 or 80286 or what the 8 means? Read the first paper to find out. The authors have over 80 years of Intel experience between them, and they are still going strong.

So what's new at Intel? The Pentium® 4 processor. Here is a short list of what makes this such an exciting product. It runs at frequencies of 1.30, 1.40 and 1.50GHz and incorporates new hyper-pipelined technology that doubles the pipeline depth to 20 stages, which significantly increases processor performance and frequency capability. Its rapid execution engine pushes the processor's arithmetic logic units to twice the core frequency, giving higher execution throughput. Moreover, it has a 400MHz system bus, Streaming SIMD Extensions 2 that extend MMX™ technology and SSE technology, and an additional 144 new instructions. The six papers in this issue, written by Intel's engineers, will give you an in-depth look at this new processor.

Also, starting with this issue you will find a new feature in the Intel Technology Journal. Each issue will have a selected list of papers published by Intel's engineers and researchers in the previous three to four months. You can use this list to find papers of interest to you. The list is located in the left top navigation bar under "Past Journals."

The Pentium® 4 Processor – Advanced Technology for the Internet and Beyond

By Ashwani Gupta
CPU Architecture Manager
Intel Corp.

The launch of a brand new microarchitecture and supporting platform, such as the Pentium® 4 processor platform, is an especially proud and exciting moment for Intel's engineers and technologists. Not only is the product launch the pinnacle of a long and intense development cycle, it is also the moment when the innovations underlying the product begin to alter the computing landscape. This allows the innovator to witness the effects of his or her ideas on the world at large.

The Pentium 4 processor platform is the beginning of a whole new family of products from Intel. The range of new technologies and innovations inherent in this platform is breathtaking and constitutes the foundation upon which Intel will be able to build for years to come. I am confident that the Pentium 4 processor will have a profound effect on the computing industry, taking performance to dizzying new heights and enabling new uses for end users. In particular, applications such as speech, natural language processing, and video are quite likely to become pervasive with the arrival of the Pentium 4 processor platform.

Several of the key innovations and technologies underlying the Pentium 4 processor-based platform are described in this issue of the *Intel® Technology Journal* by the engineers who first had the ideas and then worked long and hard to turn those ideas into reality. As you might expect, there were tremendous challenges to be overcome in the creation of these technologies. They included a very high-frequency (1.5+GHz) design with its attendant noise challenge, bucking the power trend associated with increasing clock frequencies, tuning and validating complex microarchitectures, a highly optimized and balanced system design that uses a novel chipset, a quad-pumped processor system bus and high-performance RDRAM memory, and last but not least, compiler methods to leverage new instructions introduced with the Pentium 4 processor.

The papers in this issue offer an insight into some of those challenges and how they were overcome. At the center is the new Pentium 4 processor with great performance today and enormous frequency and performance headroom for the future. At its launch frequency of 1.5GHz, the Pentium 4 processor is already in a class by itself for multimedia performance, floating-point performance, and the world's highest integer performance. This is just the beginning. As Moore's Law kicks in, and existing applications get fine-tuned for the new platform, and new applications get written that leverage the new capabilities of the Pentium 4 processor, the industry will begin to experience and appreciate the full scope and breadth of the Pentium 4 processor team's vision for this first computing platform of the 21st century.

Recollections of Early Chip Development at Intel

Andrew M. Volk, Desktop Platforms Group, Intel Corp.
Peter A. Stoll, Technology & Manufacturing Group, Intel Corp.
Paul Metrovich, Desktop Platforms Group, Intel Corp.

Index words: history, products, naming, definition, validation, debug

ABSTRACT

In the early days of Intel, between the late 1960s and the late 1970s, there was a regular product naming scheme by which a process, product type, or product family could be easily known. Few remain at Intel who remember this scheme, and its source is all but forgotten. The naming scheme and many stories of early products were uncovered through interviews and reminiscences by the authors, who among them have over 80 years of experience at Intel. This is their story.

INTRODUCTION

The genesis for this paper came from a seemingly simple inquiry to the *Intel® Technology Journal*. A reader wanted to know why "80" was used in the name of all microprocessors until the Intel® Pentium® processor. This started a search for the origins of the naming system used in the early days at Intel. It also got a few of us thinking about the early products on which we worked. In this paper, we discuss some interesting and little known facts about products introduced in Intel's first ten years, the way they were defined, developed, verified, and debugged, and how they contrast with the methods that we use today.

EARLY INTEL® PRODUCT NAMING SCHEME

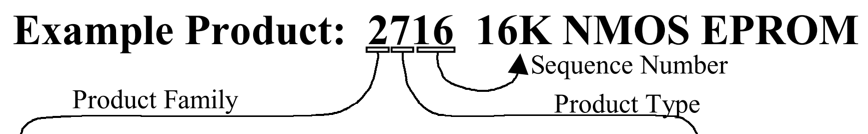
It surprised us that something as simple and mundane as the source of the early Intel® product naming scheme could be so hard to track down, but it was. In the end, we had to ask Dr. Andrew Grove, Chairman of the Board and one of the founders of Intel, for the answer. Dr. Grove said that he and Les Vadasz, then head of Engineering, worked it out one day in 1968. "I distinctly remember us concocting this scheme (minus 4XXX) sitting in his office in Mountain View, California. It worked well until marketing decided to jazz it up with 4's and 8's" [1]. Dr. Gordon Moore also was "one of the cooks" that developed the naming system [2]. So that's how it started.

Intel started with two processes: a PMOS polysilicon gate and a Schottky barrier diode bipolar process. One goal of the early products was to replace magnetic core memory in computers with silicon memories. To that end, the first products were a 64-bit bipolar memory and a 256-bit PMOS memory. The PMOS products were given numbers starting with 1xxx, and the bipolar products were given numbers starting with 3xxx. The second digit was a "1" for Random Access Memory (RAM), and the last two digits were the product sequence number. The sequence numbers of early products tended to start with "01" and went up from there. So, the first PMOS RAM was an 1101, and the first bipolar RAM was a 3101.

The 2xxx sequence started with an ambitious project to put a decoder and four 1101 RAM chips on a silicon substrate to make a 1-kilobit RAM module. The decoder was a bipolar product, the 2201, and the 2000 series was to be for hybrid products. However, the multichip module was not a success because of manufacturing difficulties and was therefore dropped. In 1971, the 2xxx sequence was given over to NMOS products.

Another form of memory was the Read-Only Memory (ROM). The first of these was a metal mask programmable 1-kilobit (256 x 4) bipolar part. The second digit "3" was assigned to ROMs. Therefore, the first bipolar ROM became the 3301, which incidentally proved to be a great source of revenue for Intel.

Intel also made shift register memory products. These were used mostly in video displays including Intel's own Microcomputer Development Systems (MDS). Intel made several early shift registers up to 1-kilobit in size. These were all dynamic memories that required that the



	Used for:	Examples:
0	Test chips	n.a.
1	PMOS products	1101*, 1103
2	NMOS products	2101, 2401, 2107B
3	Bipolar products	3101*
4	4-bit microprocessors	4004*, 4008, 4009
5	CMOS products	5101, 5201*
6	(not used)	
7	Bubble memory products	7110*
8	8-bit and beyond microprocessor and microcontrollers	8008*, 8080, 8085, 8086, 8088 8048, 8051, 8096
9	(not used)	

	Used for:	Examples:
0	Processors	4004*, 4040
1	RAMs (static, dynamic)	3101*, 2102, 2104
2	Controllers	2201, 8251, 8253
3	ROMs	3301*
4	Shift Registers†	1406*, 2401
5	EPLD†	
6	PROM	1601*
7	EPROM	1701*, 1702, 2716
8	Watch chips and timing circuits with oscillators	5801*, 5810
9	Telecommunications	2910*, 2920

* First product in this category

† There were some early exceptions. 1406/1506 were military and commercial grade shift registers, respectively. The 3404 was a latch product for memory subsystems, not a shift register.

Table 1: The Intel® product naming scheme, digit by digit

clocks be kept running. The second digits "4" and "5" were assigned to shift registers, i.e., 1402 and 1405/1505.

Programmable ROMs (PROMs) were, and continue to be, key products for Intel. Again, both bipolar and PMOS versions were developed in the early days. The bipolar parts used polysilicon fuses that were blown by pulses of high current. The PMOS memories stored charge on a floating gate. PROMs that could only be programmed once were given "6" as the second digit. The PMOS PROMs could also be erased using ultraviolet light. These erasable PROMs (EPROMs) were assigned "7" as the second digit.

The very early products were sequentially numbered. However, memory chips were soon numbered in a manner to suggest their bit size, as can be seen in the sequence of EPROM names: 2704, 2708, 2716, on up to 27512. Wanting to keep the name to no more than 5 numbers long, the 1-megabit EPROM became the 27010.

The story of the 4004 microprocessor is well known [3, 4]. The name was a marketing decision to make the 4-bit architecture clear. It wasn't an easy sell in 1971, and even in 1975 the Intel® Data Catalog introduced the Microcomputer section with two pages entitled "Why use a Microcomputer?" [5] All products associated with the 4004 were given numbers in the 4xxx sequence. Even existing products such as RAMs, ROMs, and PROMs were given 4004 family numbers, besides their normal family numbers.

In 1972, Intel acquired Microma Universal, Inc. and started in the watch business. The circuits required for these watches needed to be very low power. Consequently, a CMOS process was developed. CMOS products were assigned the "5xxx" designation. Chips that didn't have oscillators were "52xx", and chips that worked with a crystal were "58xx." Later, this CMOS process was also used for the 5101 RAM.

Also in 1972, Intel built a PMOS 8-bit microprocessor for Computer Terminals Corporation (later Datapoint). Using the same naming scheme as the 4004, this chip was the 8008. Similarly, all support chips, RAM, ROM and EPROM, for the 8008 were included in the "8xxx" family. However, the 8008 was not particularly easy to use, and a more powerful NMOS microprocessor was introduced in 1974, the 8080. This name was a simple manipulation of the same numbers. The 8080 required +12, +5, and –5 volt supplies to run. Intel also produced the three support chips that drove the 12-volt clocks and decoded the bus control signals. In 1976, a 5-volt only version that integrated the support chips was introduced. Because it required only five volts, it was dubbed the 8085. This numbering scheme continued with the 8086, introduced in 1978. Les Vadasz recalled that the name sounded good to the marketing folks as it alluded to the 16-bit architecture [6]. The expense of having a 16-bit system was reduced by the introduction of the 8088 a year later. This was a quick spin of the 8086 to reduce the external data bus to 8 bits (hence the name). IBM's choice of the 8086/88 architecture for its PC made the 8086 name extremely valuable. Subsequent processors went to 5-digit names to keep the 8086 name: 80286, 80386, and 80486. However, Intel could not get the "x86" sequence trademarked, and so the Intel Pentium processor name was born.

Because of the success of the microprocessor, the 8xxx product family has the most diverse set of products, including microcontrollers (8048, 8051, 8096) and peripheral controllers for all forms of microprocessor system functions and I/O. The first 8080 peripheral controllers were a serial I/O controller, a parallel I/O port, and a timer counter. The initial names for these chips also started out as 8201, 8202 etc., as did the early RAM chips. However, naming conflicts occurred when 3xxx family support products were renamed for use in the 8xxx family. These products were renamed 8251, 8255, and 8253 even before the designs were completed.

The last products to be assigned names were the telecommunications and analog products that used the second digit "9". The 2910 was the first single chip CODEC and was introduced in 1977. Intel also entered the bubble memory business in 1977. The "7xxx" product family was reserved for bubble memory products, and the 7110 1-megabit bubble memory chip was introduced in 1979.

And there you have it; that's how the early products were named and how the current naming scheme came about. But this is not the end of our story. Behind these product numbers are some little known histories, including some stories of products that were never in Intel's Data Catalogs. Sit back as the authors reminisce and interview other early Intel employees.

THE AUTHORS' PATHS TO INTEL

Paul Metrovich joined Intel on a bet. He was working for Union Carbide Semiconductor when that company decided to relocate to San Diego. They had subleased the building with most of the fab equipment intact to Intel. The rumor mill had it that Intel had agreed with Union Carbide not to take applications from their employees until they were ready to move their operation to San Diego. Paul bet his fellow employees $5 that the agreement did not exist. He proceeded to apply for a job, and after several interviews with Intel, he secured a position. Paul started work on April 16, 1969. He never collected on his bet.

Peter Stoll studied Electrical Engineering at MIT between 1967 and 1974, where he took several courses on circuit design, integrated circuits, and semiconductor processing. He also did a seven-month internship at Bell Telephone Laboratories working in integrated circuit design. He was not very pleased with the experience, and swore off semiconductor work when he returned to MIT for graduate school.

It did not help that the Electrical Engineering faculty at MIT in the early 1970s regarded design work with deep disdain. After a couple of years in biomedical instrumentation development, Peter realized that he didn't have the heart to pursue a multiyear Ph.D. thesis. He decided to leave school, and Intel was the only company on the interview schedule between Thanksgiving and Christmas that had compatible needs. The Intel interviewers were much more interested in Peter's design background than MIT had been and they invited him to visit. The trip resulted in two job offers. He joined in 1974 as a one-man design team designing a watch chip for Microma: the 5810.

In 1971, Andrew Volk began working with a group of students on a project to design a communication device for the handicapped called the Autocomm[1]. This project developed into his Master's thesis and involved adding the capability of typing whole words instead of letters. The design required a programmable memory to store the vocabulary. Intel had just released the 1702A EPROM and it was perfect for the job (even though storing charge on a floating gate sounded improbable to Andrew at the time). Two EPROMs could hold 64 vocabulary words (see **Figure 1**). Andrew called Les Vadasz and begged parts and technical assistance. The local sales office programmed the EPROMs and the design worked great.

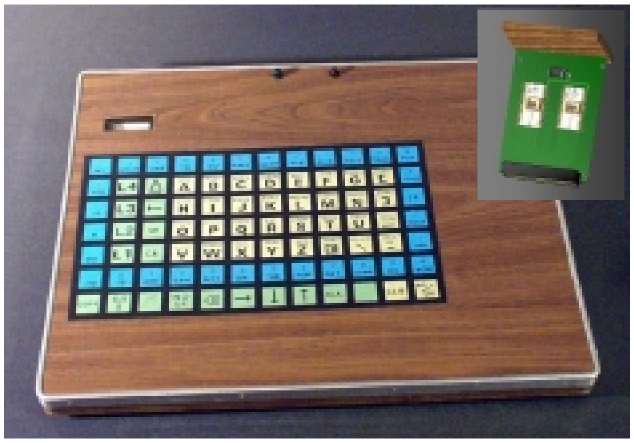


Figure 1: Autocomm and the word store using 1702A

Intel was one of the companies to which Andrew applied in 1974, and Les Vadasz came as one of the campus interviewers. He requested to see the Autocomm, which fortunately was working that day. It helped earn Andrew

[1] This group grew and became the Trace Center at the University of Wisconsin (http://trace.wisc.edu/).

a trip to California and a job offer. He started on July 1, 1974, working on the 8080A.

EARLY CUSTOM PRODUCTS

It takes time to build a market and revenue, so Intel accepted several interesting custom products in the early days. The most famous of these was the offer by Busicomm to make a 12-chip calculator chipset. Of course this led to the 4004 and microprocessor history[2]. Even the 8008 was a custom job that turned into a standard product.

Custom products also got Intel started in the dynamic RAM business. Intel worked with Honeywell on a product called the 1102 (PMOS RAM number 2). Bill Regitz was with Honeywell at the time and was hired by Intel to work on an improved part, the 1103. Everybody, including Paul Metrovich, got in on the act of trying to make this beast work reliably. Intel had a ready-made market for those parts that didn't quite meet the refresh rate specification: its Memory Systems Division. They just adjusted the refresh rate to whatever was necessary. In the end, the 1103 was a tremendous financial success.

There are plenty of lesser-known products. Tom Innes, Intel employee #38, recalls doing bipolar register and arithmetic unit chips for Burroughs Corp in 1970 (the 3405 and 3406, respectively) [7]. These were Complementary Transistor Logic (CTL) that used PNP inputs and emitter-follower outputs for high-speed and high-drive strength. Burroughs bought these chips for ten years. Ted Jenkins, Intel employee #22, started development on zinc-sulfide LEDs that emit blue light [8]. Gerry Parker, Intel employee #99, finished the work, and Intel sold it to Monsanto. We also developed a custom 7-segment decoder driver for a digital voltmeter they made. We only sold them 10,000 devices, a very small number in our business.

In 1972, Intel's EPROM technology attracted the interest of Mars Money Systems (MMS) who wanted a chip for an electronic coin changer. MMS was a wholly owned subsidiary of Mars, Inc., the candy and food products company. Mars had gotten into the vending business quite early as a means to distribute their product. Accurate coin handling was critical to getting good revenue return as well as customer satisfaction. However, a good coin changer was a real Rube Goldberg[3] contraption of delicately balanced levers and magnets.

[2] See this history and others at the Intel Museum. Visit it on-line at http://www.intel.com/intel/intelis/museum/.

[3] For those too young to know who Rube Goldberg is, see the web page at http://www.rube-goldberg.com/.

Fred Heiman, President of MMS at the time, invented an electronic means of differentiating coins using tuned coils. Using this scheme, Intel developed the 1205 and 1206 chips for MMS. We know the part number only because Paul Metrovich kept one as a souvenir in his toolbox. Paul worked on a prototype of discrete parts that proved the concept was feasible. Mr. Heiman recalls that it took less than one year to get it working and required about 3,000 transistors. He said that it worked wonderfully and had a product life of about five to six years. A coin reject solenoid was the only moving part in the coin mechanism [9].

Because the 1205/06 chip had an erasable PROM, it was self-calibrating. A replacement coin detector coil did not necessarily react the same as the previous one. The 1205/06 could be erased with ultraviolet light and a set of calibration coins fed through the coin changer to set the limits of detection. The results were programmed into the device while still in the vending machine. When new slugs were detected, their characteristics could be studied and new calibration coins developed to exclude them. MMS is now Mars Electronics, Inc. and still a large player in vending and coin, and in bill changing.

(Forest Mars, Sr., retiring head of Mars, Inc., visited Mr. Heiman about one year after the electronic coin mechanism went into production to understand it and its capabilities. He asked Mr. Heiman to arrange a meeting with the head of Intel and a meeting was set with Dr. Noyce a week later. He sat and listened to Bob talk about how Intel was growing and innovating on this "crest of technology." That was enough for Mr. Mars to decide that he had no interest in buying Intel. He was used to developing long-term products with steadier sales than these new silicon devices that Intel was creating. Mr. Heiman noted, "Perhaps if the pace of silicon technology was a little slower, Intel might have become a division of Mars, Inc" [10]. No one at Intel was aware of this possibility, and as Les Vadasz noted "…we were not for sale, anyway" [11].)

One of Paul's favorite custom parts was the 8244. It was a TV game chip that, when coupled with an 8048 microcontroller and a ROM, became the Magnavox "Odyssey 2." It had a great nine holes of golf! Intel made good money on it. There was also an 8245 chip for European PAL television that differed from the 8244 only in the number of scan lines per frame and the timing of the TV sync outputs.

PRODUCT DEVELOPMENT IN THE DARK AGES

If we compare the tools we had available to us 25-30 years ago to the tools we have today, we would definitely call that period the dark ages of silicon development. The steps for developing a chip back then and now are much the same in a broad sense: definition, logic and circuit design, verification, layout and mask making, silicon wafer fabrication (processing), and debug and test. But that is where the similarity ends. In the early days, design, verification, and testing were done manually for the most part. Fortunately, the chip designs back then contained fewer than 30,000 transistors instead of today's 42 million.

Today, chip definitions require specifications hundreds of pages long, logic design is largely a matter of writing software code, computers run millions of verification tests on logic and timing in a few days, and testing is done on multi-million dollar testers. This section describes some of our experiences with early chip development.

Product Definition

When Peter joined Intel in 1974, he was the sole design engineer on the 5810. The product definition process for that chip illustrates a radical difference between the Intel of then and the Intel of today. His boss, Joe Friedrich, prepared a single page document called a Target Specification (spec) that gave the four-digit name to the product. It also gave the pinout and defined the function in sufficient detail for the approving parties to decide whether they wanted to build it. It described to Peter what he had to build.

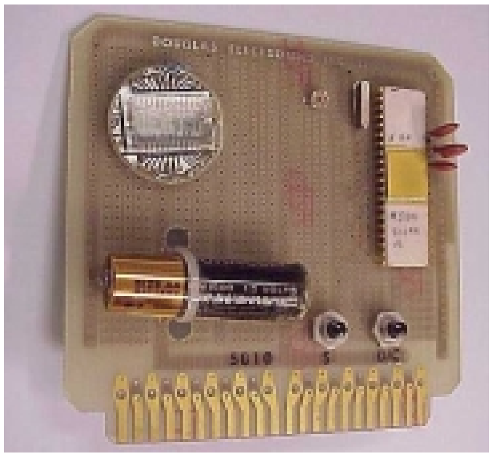


Figure 2: Peter Stoll's prototype 5810 "watch"

The entire chain of command of Intel, from Robert Noyce on down to Joe Friedrich, met in a room to decide whether to approve development based on the 5810 Target Spec. In that single meeting, the decision was made to proceed. The product name, 5810, remained constant from that point forward throughout the product life. The name appeared in the Target Spec, the schematics, any memos, the actual layout, the masks, the marketing printed materials, fab lot yield reports, and anywhere else the part was discussed.

The initial 8085 Target Spec was also very simple since we were integrating the functions of the 8080 with its clock and system controller. Only a simple serial I/O and some additional interrupts were added. It took only two pages.

After the project got going, several attempts were made to change the product, especially in light of rumors of a product from Zilog (the Z80). There was an attempt by our manager to make it into a micro-VAX. Eventually, he gave up on the 8085 and turned his attention to the next chip, the 8086.

The simplicity of the early decision process and nomenclature stands in stark contrast to our practices even in 1978. By that time, product definition took months, engaged many committees, created multiple distinct fat memoranda, and generally frustrated all involved to no end. Also, by 1978 we had started our current practice of confusing ourselves by referring to the exact same product by many (and often changing) code names. Certainly, the complexity of today's products requires more complete documentation, but we've also made the job harder by not following some of the simple rules of nomenclature we followed in earlier, simpler times.

Logic and Circuit Design

There were no logic design tools when the authors started at Intel, no VHDL or logic synthesis. The gate-level design we learned in school was replaced by transistor-level design in order to get the most efficient transistor counts and the smallest layout area. Repeating functions were designed as cells, but the cell was still optimized at the transistor level.

About the only computer design tool we used in 1974 was an in-house analog circuit simulation tool called SPULS. In contrast to today's highly sophisticated and heavily constrained computer design tool environment, a new design engineer's entire training on our computer tools took about half an hour. We were shown the common terminal area, which consisted of a short row of dumb terminals connected to the one central PDP-10. By the end of the half-hour we knew how to log in and how to run the simple text editor. We could specify a circuit of five to a few dozen transistors and tell the circuit simulator what input signals should be simulated and what output signals should be monitored. The result was provided as "line printer graphics" with a resolution in both time and voltage of whole character cells. The y-axis was limited to 70 or 120 points (characters) depending on the printer's carriage width.

In 1975-1977, when all of the original circuit design for the 8085 and 8086 microprocessors was carried out, the circuit size our central computer could handle was so small that we never simulated complete circuits or entire circuit paths. The circuit was decomposed into small pieces of about 5-20 transistors, simulated, and then added manually back together based on our understanding of the overall subsystem. The simulator was necessary for circuits such as RAM sense amplifiers, input buffers, and internal precharge-discharge buses.

The circuit size was limited by computing constraints such as memory. Another equally important size limit was the mean time to the computer crashing, which happened as often as every 15 minutes. If the computer went down, we lost the whole run. This also applied to file editing. There were no auto-backup files. We learned by brutal experience to save our work frequently.

There was, of course, no computer tool to extract parasitic capacitances from the actual layout, so accuracy in speed simulation was largely dependent on the design engineer's skill in guessing layout distances and routings.

Large portions of the logic circuitry of both the 8085 and 8086, as with other microprocessors and controller parts at that time, were composed of simple n-channel, depletion-load logic. On the 8085, Peter constructed a table estimating the delay for each size of depletion load transistor we used versus various circuit loads. This “paper computer” was used in place of circuit simulations for the overwhelming majority of the speed paths. The errors from these tables were quite small when the layout parasitic estimations were done reasonably well.

Breadboards and Prototypes

Since simulation was limited, many other means were used to verify parts and new ideas. Paul remembers that the PMOS EPROMs were first prototyped by Dov Frohman, inventor of the EPROM, using a 4x4 array of discrete transistors in TO-5 packages on a special breadboard to enable programming and reading. A similar 16-bit array was put on to the first 1701, but since the full 256x8 array worked, the small array was never really tested [12].

Quite a few parts, ranging from the 1850-transistor 5810 watch chip, up to at least the 6144-transistor 8085 microprocessor, used no logic verification technique other than the engineer's brain. Andrew spent weeks in 1976 playing “computer” by running through all the 8085 instructions.

Several other development projects did construct a prototype breadboard, typically using commercial logic components such as 7400-series TTL to reproduce the logic proposed for the chip. It was always difficult to get the breadboard done before the part was ready to tape out. Also, there were never commercial components available to reproduce all the functions we used on the chips. Breadboards were valuable to debug designs, and they provided a pre-silicon device to check the tester functionality. It was also valuable to check factors not easily seen on a simulation. We liked to use prototypes for human interface devices, like video displays or games.

Breadboarding was feasible until product device counts numbered in the tens of thousands of transistors. Eventually, the breadboard became too large and complicated to keep up with the speed of the real silicon product. The last custom breadboard Andrew and Paul constructed was a video terminal device, one of the first 5-digit (82730) part numbers in the early 1980s.

Ironically, a new form of breadboarding called emulation is being used now to verify chip designs with millions of transistors. The chips’ functions can be programmed by software into the emulator instead of having to solder or wirewrap discrete logic. Now we are able to essentially boot the PC without having to build any chips.

Logic Simulation

Intel's first in-house logic simulator was LOLA/LOCIS, developed by a team headed by Mark Flomenhoft. It became ready for first use just in time to be used on the 8086 microprocessor project. Our use of this tool on the 8086 helped us find dozens of logic errors before the first stepping was taped out (although we did leave a few more to find in the actual silicon!). A parallel breadboard project consumed at least five times the staff, quite a bit more equipment, money, and lab space than the logic simulation effort, but the logic simulation effort found more problems sooner. (Jim McKevitt, lead designer on the 8086, found at least as many bugs using no tools other than his brain, the schematics, and a large supply of well-sharpened pencils.)

Layout and Mask Making

Schematic and layout for the first ten years of Intel was done by hand. Engineers would produce draft schematics that a schematic designer would transfer onto D-sized vellum sheets. These would then be hand checked and signed off by the engineer. All edits to the schematic would be noted, checked, and signed off.

Layout planning was done between the engineers and the layout designer (mask designers). The layout of the 8085 was easier than most chips since it followed the base floorplan of the 8080. Peter guided most of the layout work, while Andrew did the layout of the control logic array. This was a ROM-like array based on a dual sum-of-products structure. Andrew planned it out on graph paper, carefully folding the terms together to meet layout constraints while still minimizing the size. It took two weeks to get the final layout plan. (Andrew still has those planning sheets.)

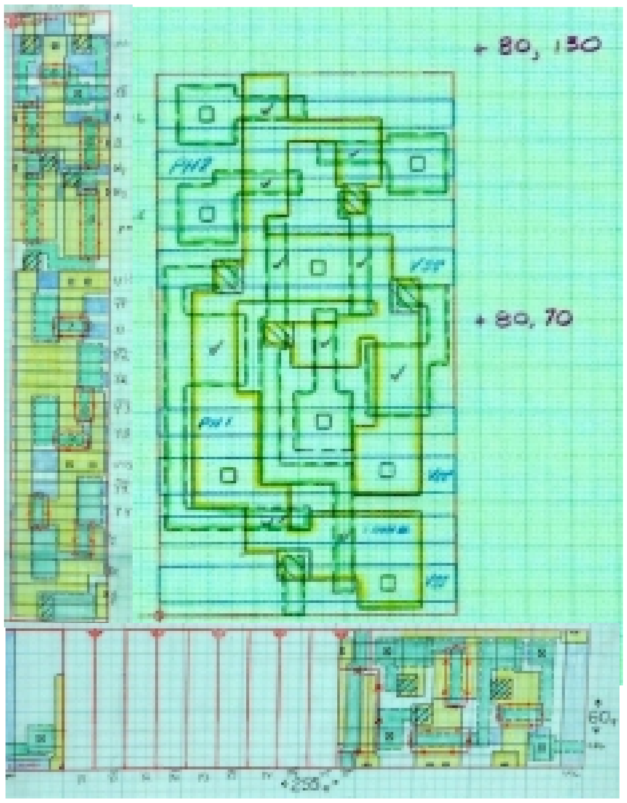


Figure 3: Hand-drawn cell layout on Mylar

There was absolutely no computer assistance for design rule verification or for logic vs. layout wiring correctness. Physical layout proceeded as highly skilled mask designers drew lines with pencils on very large sheets of gridded Mylar (**Figure 3**). By 1974, the result was being digitized on a Calma GDS I system so repeated cells could be handled automatically, instead of being hand drawn every time. But the crucial questions of whether the drawn lines actually represented the same circuit called for by the schematics, and also whether the drawn lines honored the design rules, were entirely governed by human diligence. Even after thoroughly checking the layout, the most skilled of our mask designers left quite a few errors in their initial work. Finding and removing all errors was a very difficult part of the work.

We often built our own aids to try to make design rule verification go a bit more efficiently. Peter drew concentric square boxes on translucent Mylar as a visual aid for design rule checking. He moved his drawing around to every single contact drawn on the chip, trying to find violations of the rules governing widths and spaces around contacts.

The authors believe that most chips in those days shipped with at least some design rule violations. But you really couldn't expect the part to work if it was not wired up correctly. So in addition to daily comparisons of the schematics to the drawn layout, a lot of energy went into a final check before digitizing and another before tape out. Our usual practice was to start with a full schematic of the entire chip, a yellow pencil, and a dark pencil. As we matched up layout found on the plot created from the digitized artwork with the schematic, we would mark the matched circuits in yellow on the schematic and write in signal names on the plot. We were still doing it this way for the 8086 first stepping in 1977. That part had 20,000 transistors, and it took two weeks for each of the two design engineers who performed the final task. Both engineers (Peter and Chun-Kit Ng) found 19 of the same 20 errors, which was considered quite a good detection rate for this particular technique. A few months later, Todd Wagner provided Intel's first logic vs. layout connectivity verification tool, which relieved future generations of design engineers of this onerous task.

The first masks were made by transferring the drawings on the Mylar to "rubylith." Rubylith is a two-layered material, which comes in huge sheets. The base layer is heavy transparent dimensionally stable Mylar. A thin film of deep red cellophane-like material covers the base layer. The first chips at Intel used a machine called a "Coordinatograph" to guide cutting of the ruby layer. The coordinates and lengths had to be measured and transferred by hand to the cutter. Later, a Xynetics plotter with knives, instead of pens, was used to cut more quickly and precisely.

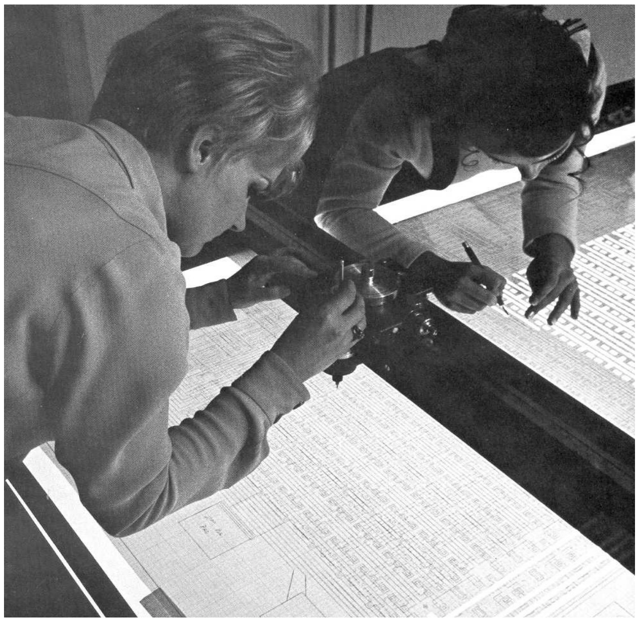


Figure 4: Technicians transferring layout to rubylith

When the cutting was finished, the technicians had to peel away only the desired geometries that made the mask layers. The design engineer and mask designers would spend days hand-checking the rubylith for peeling errors, nicks, and unintended cuts. A final check was made for design rule violations. The rubylith was sent to the mask vendor to be made into masks for fabricating the silicon die.

Missing a cut or forgetting to peel a geometry would mean a bad part. Ted Jenkins remembers working on the first Intel product, the 3101 64-bit RAM. Actually, the first version was only a 63-bit RAM due to a simple error peeling one layer on the rubylith [8].

The rubylith sheets had to be handled very carefully so they were not damaged. Small areas of ruby could be rubbed off. Andrew remembers a call from the 8080A mask vendor saying that they had found a "floater," an unexplained piece of ruby stuck in a random place on the Mylar. They feared that a piece had come off somewhere. A several hour check against the layout found no missing bits and the mask was taken as is. Fortunately, the dice made with that mask were okay.

Adding or removing transistors and interconnect on rubylith was definitely a manual task, not unlike surgery. In fact, the technician who did the edits used a surgical scalpel and a metal ruler (scale). Adding transistors or interconnect involved cutting and peeling away bits of ruby. Removing objects involved adding ruby-red tape to the back of the heavy Mylar. Cuts had to be precise so as to leave no nicks or cut marks on the Mylar that might show on the mask. Verification was done with the metal scale and a 7X-magnifying eyepiece with a calibrated scale on the bottom.

Processing

Ted Jenkins was responsible for developing Intel's CMOS process to support the watch business. Intel needed ion implantation for CMOS, but didn't have the equipment. So, the first wafers were made at Extrion (since acquired by Varian). The process was ready before the first timing chip designs were ready.

The first P-MOS PROMs were in packages with metal lids and could not be erased with ultraviolet (UV) light. It was suggested that perhaps X-rays could be used and this was tried. It was unsuccessful for two reasons. It took a lot of X-rays to erase the memory properly and when the process was complete, the X-rays had damaged the transistors, permanently changing their electrical characteristics.

Customers were skeptical of the reliability of the early EPROMs and were afraid that sunlight would erase them. To test the technology, 1702s were left on the roof of an Intel® building in full sunlight for many days with no data loss. (Later N-MOS EPROMs were, in fact, more sensitive to ambient UV, so a yellow tape was applied to the quartz lid to block the UV. The tape was removed for erasure and reapplied for use.)

Tom Innes recalls an attempt to make a bipolar PROM with floating gates! [7] A P-channel floating gate device was inserted in the base of a PNP transistor, and it was programmed by breaking down the collector-base junction. The oxides were not good though and the retention was from a few weeks at best to hours at worst. Jean-Claude Cornet and Fred Tsang, early Intel employees responsible for bipolar product development, came up with the poly fuse concept that was used for bipolar PROMs.

The 8085, 8086, and SRAMs used the same NMOS processes. In the mid-70s, the SRAM business was seen as a larger revenue source than the microprocessors. Tweaks were made to the process to improve SRAM performance without worrying about the impact on the microprocessors. Today, it would be strange to think that an SRAM process requirement was more important than a microprocessor design.

A bit later, Intel developed its dual implant NMOS process called "HMOS" for high-speed SRAMs. These SRAMs were replacements for bipolar RAMs being offered by a few competitors. Our parts were just as fast (15 ns access time), but were much cheaper to build and consumed a fraction of the power. One normally quiet and reserved process engineer designed a T-shirt with appropriate graphics and the slogan: "Cure your blazing Bipolar itch with Preparation HMOS!"

HMOS was a very robust process. The first SRAMs made with HMOS were packaged in a white ceramic package. The parts were tested for reliability with a "life test" of 1000 hours in a burn-in oven at 125°C. For one batch of parts, the oven temperature control failed and got to over twice that temperature before being shut down. When the burn-in boards were removed, the sockets holding the parts had melted down the boards like wax. The parts themselves were the color of toasted brown marshmallows. Incredibly, the vast majority of parts survived quite well with little impact on their performance.

Test and Debug

In the process of developing DRAMs, it became apparent that there was a need for specialized test equipment. Initially, engineers used simple switch boxes and fixtures with signal generators and viewed results on an oscilloscope. The wafer prober was operated by hand and bad dice were marked with a felt-tip pen. However, this arrangement soon proved too tedious, and commercial LSI test systems were purchased. These were rudimentary machines that came with a high price tag.

Paul was chartered with the task of designing, building, and operating an engineering-level LSI memory tester for the MOS design team. The first product to be tested on this unit was the 2107 4096 bit dynamic RAM, still in design. He was given the substantial budget of $165,000 (quite large for a starting company) and some technical and assembly people to help. The result was a rack with lots of controls, and a central changeable fixture for different types of devices.

Paul dubbed the machine the Tel-Tester. He started it in the Mountain View facility and completed it in the Fall of 1971 after moving into the first site owned by Intel in Santa Clara. The system was designed with Emitter-Coupled Logic (ECL) allowing a basic clock of 100MHz to be used to time the unit. The test system was unique in several ways. Digital switches controlled the timing and voltage levels. It also had an interface with automated wafer-probing equipment, allowing sorting of pilot runs of engineering-level memory products. An added feature was a built-in oscilloscope with a raster scan display of the memory array under test with errors or data patterns highlighted for analysis. Some thought was given to a computer interface, but it was not implemented due to cost and time constraints.

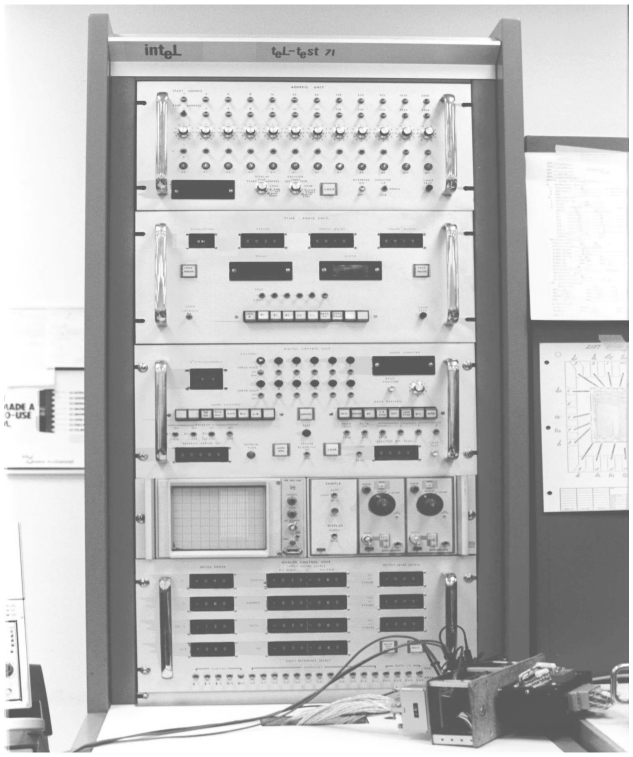


Figure 5: The Tel-Tester for checking DRAMs

The Tel-Tester served well in the lab, lasting through several generations of DRAMs. Others used it for several more years in the memory products groups until lower-cost commercial memory testers became available.

Paul moved on to the new microcomputer group that was designing the 4004 and other computer system-related devices. He was engaged in building breadboards of products and providing a new way to test these devices. We had neither the luxury of a long time nor a large budget to develop bench test equipment, and he had to find a faster, cheaper way to meet the needs of test.

Eventually, the idea came to Paul to make a standardized desktop tester to evaluate and do design verification for new products. It was called a Modular Test System or MTS box, but was better known as a T-box.

Paul built these T-boxes from a standard metal chassis and included a standard power supply module, an opens and shorts parametric module, a matrix switching module, and an open space for multiple custom boards to do some functional testing on whatever product the system was targeted to test. An MCS-4 microcomputer module with a 4004 was used for the control system.

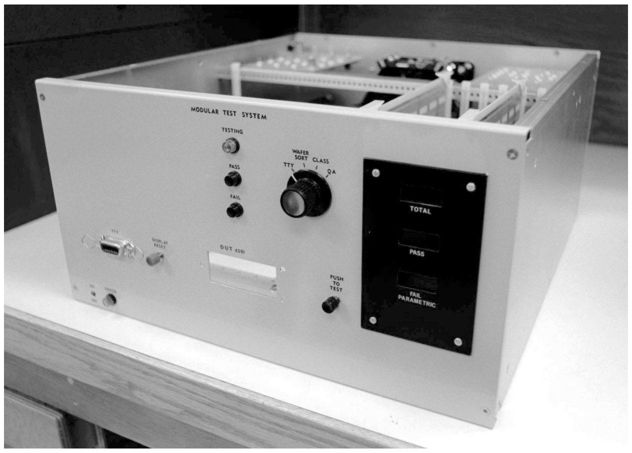


Figure 6: Paul Metrovich's benchtop MDS test box

Several boxes were built during slow times. The custom test module was designed when a new product was ready. The parametric module was programmable as to which device pins were to be tested and which were not.

Additionally, it was decided to provide a T-box to the production test group with every new logic type device. (Eventually, a group of engineers, technicians, and assemblers were formed to do this, first with T-boxes and later with the functional test modules of purchased testers. This became Intel's Test Engineering group.)

At first, the boxes were serialized, but they ended up being named after the part number of the device to be tested. For example, the tester for the 8080 was the T-80. A simplified version of the T-box was built just to do an opens/shorts test on products after assembly from wafers.

Another test concept was doing comparison functional testing between a "golden" device and the Device Under Test (DUT). This was a good idea, but it became a chicken-and-egg contest to find a device that was, in fact, golden against which all other devices could be tested.

Some of the "golden" device comparison testers presented real technical challenges due to the uncertainties of synchronizing various clocks and data simultaneously in both the DUT and the "golden" device. A case in point was the 8251 USART. The data word was supposed to be aligned when the parallel data were written into both parts. However, there was a timing variation of up to 8 clocks before it came out the serial data port of each device, which messed up direct comparisons. (Besides this, the earliest version of the 8251 USART chips had a quirk in them. Millions of bytes were written in and occasionally one byte would never come out the serial port due to a bad internal voltage level!)

The whole thing culminated in the fact that Intel was not really interested in being in the test equipment business. We needed the units, but could not purchase them, and the large LSI testers were still in the design stages of development. Intel was always ahead of the support marketplace.

Steve Bissett, Andrew's early mentor on the 8080A, was working on getting the 8080A tested on the T-80. The T-80 was not very reliable, and multiple passes of the same set of parts would yield quite variable results. This led Steve to believe there was a better way to test. He seized the opportunity by leaving Intel and founding MegaTest. He designed the MegaTest Q8K test box, a machine similar to the MTS but with refinements. Intel bought quite a few.

(One story Andrew will never forget was the day he asked Steve what the 8080A die looked like. They were selling for $360 each in those days. It was packaged in a ceramic package with a gold-plated lid. "Steve selected one of the parts he was testing on the bench, dropped it to the floor and stepped on it, cracking open the package. As he picked up the part and pulled it apart to show me the die, all I could think was $360! He just stepped on $360! That was a good chunk of my paycheck then.")

Peter remembers the test setup commonly built to check the functionality of initial samples of the product, and even the testing of initial samples for shipment to customers. For a watch chip, this generally meant arranging a probe card to actually probe the dice on the wafer, an interface cable, a watch display (LCD or LED), a few switches, and power. Then the engineer for the part, or a technician, would sit for endless hours at the lab bench, flipping switches and watching the display, deciding whether each die appeared to work or not.

Peter dreaded the prospect of spending weeks flipping switches, and doubted the resulting product quality. He spent several of the weeks between making masks and getting the first wafers designing and building a small informal tester. It checked whether the on-chip voltage tripler could actually generate the required power supply voltage, supplied an extremely simple set of input signals to the watch chip, and checked whether the outputs were correct by comparison to a known good reference. He even added logic to mark each bad die and automatically step the probe card across the wafer. He still had to flip switches and look at the display for the very first chip he tested, and it worked. Peter manually tested about 10 more chips, but after that, the improvised tester was good enough to determine initial yield and to create initial customer samples. The 5810 proved to be production worthy on the first stepping of the die. Packaged parts from this first lot were also provided to the T-box developer to allow him to carry out tester development.

Peter got a bit of help in assembling his informal tester from one of the lab technicians, but he did nearly the entire chip checkout, evaluation, and sample generation himself. In those days, a product design engineer could expect to be heavily involved in nearly all phases of product development. For many of us, this relationship created a deep satisfaction and an intense sense of ownership.

Peter and Andrew also designed a test setup for the 8085 chip that was, in essence, a small computer system where the 8085 under test actually executed its own test program. This was a dangerous strategy since the part needed some functionality to even get the test started. Both the part and tester worked well enough to allow us to debug the chip from the start (after making an allowance for an inversion on the chip's address bus tri-state control).

The dedication we all felt to the products can be demonstrated by a final story. The stepping of the 8085 that was expected to allow volume production of the part came out of the fab over a week earlier than expected, on October 21, 1976 to be precise. Andrew remembers that date well. It was the day before his wedding. But he still stayed until midnight checking out the new stepping. He left a short report on his boss' chair saying that all the bugs found in the previous stepping were checked and working, and that he would do a more thorough evaluation—in a week!

CONCLUSION

It is hard to end this story. Each time a name or event is mentioned, it triggers yet another episode buried somewhere in our memories. It has allowed us to briefly revisit a time when we were heavily involved mentally, physically, and emotionally in our work. It was a time when we felt we were entirely responsible for a project. Writing this has stirred feelings that have lain dormant for a long time, yet come flooding back upon hearing the stories, seeing the pictures, and talking with past colleagues.

There are many more articles and histories floating around in the minds of the good employees, present and past, who contributed to building and sustaining the corporation called Intel. Each individual has a story to tell, a joke to make you laugh, an incident to relate that evokes a touch of anger, and a personal anecdote that makes a career a life experience. To discover and recount all of these would take another lifetime and result in a large book rather than a journal article. For now, we just want to record some of our early experiences in a young corporation by highlighting how products were named and developed in the first years. We hope we have done this in a way that brings across the fun we had, the effort we put in, and the results we achieved.

ACKNOWLEDGEMENTS

It was a real challenge to find people who were around during the first years of Intel Corporation to check the information in this article. To those who contributed to this paper, thank you for sharing your life work and memories. Of particular assistance were Dr. Gordon Moore, Tom Innes, Ted Jenkins, and Les Vadasz who provided additional information and reviewed our article. We also got a final verification of the naming system from Dr. Andrew Grove (who was definitely here from the beginning). Dov Frohman provided insight on EPROM development and Mr. Fred Heiman gave us the inside story on the 1205 Mars Money Systems chip. Thanks also go to Tracey Mazur of the Intel® Museum for help with reference material. Peggy Thompson and Norman Strand get honors for still having examples of hand-drawn layout, the most having been long since scrapped. We also thank the Trace Center for providing photos of an early Autocomm.

REFERENCES

[1] Grove, Andrew, "Question on history of product numbering," E-mail to Andrew Volk, Dec. 8, 2000.

[2] Moore, Gordon, telephone interview, Jan. 16, 2001.

[3] Noyce, R. and Hoff, T., "A History of Microprocessor Development at Intel," *IEEE Micro*, Vol.1, No. 1, Feb. 1981, pp. 8-11, and 13-21.

[4] Freiberger, P. and Swaine, P., *Fire in the Valley – The Making of the Personal Computer, Second Edition,* McGraw-Hill, New York, NY, 2000, pp. 15-23.

[5] Intel Corporation, *1975 Data Catalog*, Sect. 6, pp. 3-4.

[6] Vadasz, Les, telephone interview, Dec. 4, 2000.

[7] Innes, Tom, telephone interview, Dec. 8, 2000.

[8] Jenkins, Ted, telephone interview, Dec. 7, 2000.

[9] Heiman, Fred, telephone interview, Dec. 5, 2000.

[10] Heiman, Fred, "Interview notes on Mars chip," E-mail to Andrew Volk, Dec. 7, 2000.

[11] Vadasz, Les, review comments on this paper for the Q1 *Intel Technology Journal*, quoted in an e-mail to Andrew Volk, Dec. 19, 2000.

[12] Frohman, Dov, "Search for old information," e-mail to Paul Metrovich, Dec. 8, 2000.

AUTHORS' BIOGRAPHIES

Andrew Volk is a 26-year veteran of Intel. He started his design work on the 8080A. He was responsible for the logic design on the 8085 and has been lead designer and project manager for several I/O products for embedded control, bubble memories, and personal computers. Currently, he is a Principal Engineer responsible for developing special circuits especially as they apply to new technologies. He specializes in circuit design and physical interface definition for various buses. He was responsible for the electrical bus definition for the Universal Serial Bus (USB), Accelerated Graphics Port (AGP), and the link interface for the Intel® Hub Architecture chipsets. Andrew is the holder of 19 patents with 23 pending, has written several technical articles and a chapter in a computer technology book, and has been a speaker at industry conferences. He joined Intel after receiving his Master's degree in Electrical and Computer Engineering from the University of Wisconsin. He enjoys reading, walking, fishing, and travel. His e-mail is andrew.m.volk@intel.com.

Peter A. Stoll first joined Intel in 1974. He also joined Intel in 1978 and 1987! Peter's design engineering work included a watch chip, the circuit design of the 8085 microprocessor, and circuit design, logic simulation, and speedup stepping work on the 8086 microprocessor; and, a few days devising and implementing an emergency fix to the 80386 32-bit multiply problem. Peter also worked at Hewlett-Packard in high-speed GaAs logic circuit design. At Daisy Systems he was the primary developer of a hardware/software product allowing use of logic parts as their own models in logic simulations. Since 1988, Peter has worked in the Intel® Albuquerque wafer fabs in reliability, yield, and product engineering roles. He has worked most frequently on the large-scale use of available technical data from wafer and unit production to help resolve yield and reliability problems. Peter holds SB, SM, and EE degrees from the Massachusetts Institute of Technology, where he was admitted as a Presidential Scholar, and he was admitted to Eta Kappa Nu, Sigma Xi, and Tau Beta Pi honoraries. His e-mail is peter.a.stoll@intel.com.

Paul Metrovich joined Intel on April 16, 1969, working in the Design Engineering department in the original Intel® site in Mountain View, California. He worked on the first dynamic shift registers and the 1103 DRAM. He also designed and built internal engineering test equipment (MTS) for DRAMs and early microprocessors, emulation breadboards of microprocessor support devices, and custom chips. Paul assisted with product design validation on such products as the 8251 and other system products based on the 8080/8085, along with magnetic bubble memory support products and video games. He moved to Folsom, California in 1985 with the Peripherals group, assisting with debug and validation on floppy disk controllers and the PC chipset devices. Paul currently is the lab manager, designing and building specialty labs for the Chipset Engineering and Validation group. He received his electronics education in the U.S. Navy, and he has had prior employment at Varian Associates and Union Carbide Corp. His e-mail is paul.t.metrovich@intel.com.

The Microarchitecture of the Pentium® 4 Processor

Glenn Hinton, Desktop Platforms Group, Intel Corp.
Dave Sager, Desktop Platforms Group, Intel Corp.
Mike Upton, Desktop Platforms Group, Intel Corp.
Darrell Boggs, Desktop Platforms Group, Intel Corp.
Doug Carmean, Desktop Platforms Group, Intel Corp.
Alan Kyker, Desktop Platforms Group, Intel Corp.
Patrice Roussel, Desktop Platforms Group, Intel Corp.

Index words: Pentium® 4 processor, NetBurst™ microarchitecture, Trace Cache, double-pumped ALU, deep pipelining

ABSTRACT

This paper describes the Intel® NetBurst™ microarchitecture of Intel's new flagship Pentium® 4 processor. This microarchitecture is the basis of a new family of processors from Intel starting with the Pentium 4 processor. The Pentium 4 processor provides a substantial performance gain for many key application areas where the end user can truly appreciate the difference.

In this paper we describe the main features and functions of the NetBurst microarchitecture. We present the front-end of the machine, including its new form of instruction cache called the Execution Trace Cache. We also describe the out-of-order execution engine, including the extremely low latency double-pumped Arithmetic Logic Unit (ALU) that runs at 3GHz. We also discuss the memory subsystem, including the very low latency Level 1 data cache that is accessed in just two clock cycles. We then touch on some of the key features that allow the Pentium 4 processor to have outstanding floating-point and multi-media performance. We provide some key performance numbers for this processor, comparing it to the Pentium® III processor.

INTRODUCTION

The Pentium 4 processor is Intel's new flagship microprocessor that was introduced at 1.5GHz in November of 2000. It implements the new Intel NetBurst microarchitecture that features significantly higher clock rates and world-class performance. It includes several important new features and innovations that will allow the Intel Pentium 4 processor to deliver industry-leading performance for the next several years. This paper provides an in-depth examination of the features and functions of the Intel NetBurst microarchitecture.

The Pentium 4 processor is designed to deliver performance across applications where end users can truly appreciate and experience its performance. For example, it allows a much better user experience in areas such as Internet audio and streaming video, image processing, video content creation, speech recognition, 3D applications and games, multi-media, and multi-tasking user environments. The Pentium 4 processor enables real-time MPEG2 video encoding and near real-time MPEG4 encoding, allowing efficient video editing and video conferencing. It delivers world-class performance on 3D applications and games, such as Quake 3[*], enabling a new level of realism and visual quality to 3D applications.

The Pentium 4 processor has 42 million transistors implemented on Intel's 0.18u CMOS process, with six levels of aluminum interconnect. It has a die size of 217 mm^2 and it consumes 55 watts of power at 1.5GHz. Its 3.2 GB/second system bus helps provide the high data bandwidths needed to supply data to today's and tomorrow's demanding applications. It adds 144 new 128-bit Single Instruction Multiple Data (SIMD) instructions called SSE2 (Streaming SIMD Extension 2) that improve performance for multi-media, content creation, scientific, and engineering applications.

[*]Other brands and names are the property of their respective owners.

OVERVIEW OF THE NETBURST™ MICROARCHITECTURE

A fast processor requires balancing and tuning of many microarchitectural features that compete for processor die cost and for design and validation efforts. Figure 1 shows the basic Intel NetBurst microarchitecture of the Pentium 4 processor. As you can see, there are four main sections: the in-order front end, the out-of-order execution engine, the integer and floating-point execution units, and the memory subsystem.

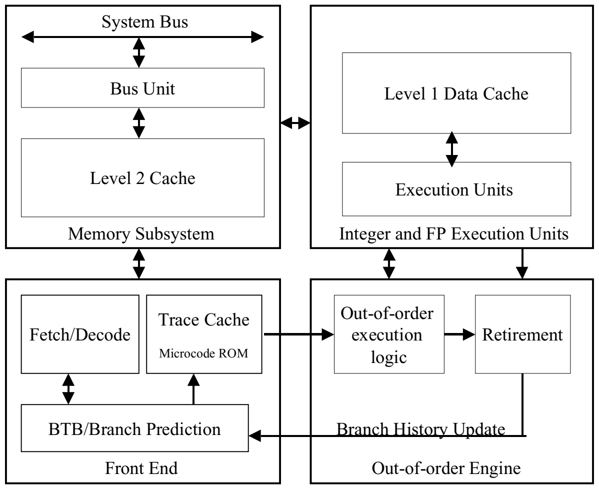


Figure 1: Basic block diagram

In-Order Front End

The in-order front end is the part of the machine that fetches the instructions to be executed next in the program and prepares them to be used later in the machine pipeline. Its job is to supply a high-bandwidth stream of decoded instructions to the out-of-order execution core, which will do the actual completion of the instructions. The front end has highly accurate branch prediction logic that uses the past history of program execution to speculate where the program is going to execute next. The predicted instruction address, from this front-end branch prediction logic, is used to fetch instruction bytes from the Level 2 (L2) cache. These IA-32 instruction bytes are then decoded into basic operations called uops (micro-operations) that the execution core is able to execute.

The NetBurst microarchitecture has an advanced form of a Level 1 (L1) instruction cache called the Execution Trace Cache. Unlike conventional instruction caches, the Trace Cache sits between the instruction decode logic and the execution core as shown in Figure 1. In this location the Trace Cache is able to store the already decoded IA-32 instructions or uops. Storing already decoded instructions removes the IA-32 decoding from the main execution loop. Typically the instructions are decoded once and placed in the Trace Cache and then used repeatedly from there like a normal instruction cache on previous machines. The IA-32 instruction decoder is only used when the machine misses the Trace Cache and needs to go to the L2 cache to get and decode new IA-32 instruction bytes.

Out-of-Order Execution Logic

The out-of-order execution engine is where the instructions are prepared for execution. The out-of-order execution logic has several buffers that it uses to smooth and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. Instructions are aggressively re-ordered to allow them to execute as quickly as their input operands are ready. This out-of-order execution allows instructions in the program following delayed instructions to proceed around them as long as they do not depend on those delayed instructions. Out-of-order execution allows the execution resources such as the ALUs and the cache to be kept as busy as possible executing independent instructions that are ready to execute.

The retirement logic is what reorders the instructions, executed in an out-of-order manner, back to the original program order. This retirement logic receives the completion status of the executed instructions from the execution units and processes the results so that the proper architectural state is committed (or retired) according to the program order. The Pentium 4 processor can retire up to three uops per clock cycle. This retirement logic ensures that exceptions occur only if the operation causing the exception is the oldest, non-retired operation in the machine. This logic also reports branch history information to the branch predictors at the front end of the machine so they can train with the latest known-good branch-history information.

Integer and Floating-Point Execution Units

The execution units are where the instructions are actually executed. This section includes the register files that store the integer and floating-point data operand values that the instructions need to execute. The execution units include several types of integer and floating-point execution units that compute the results and also the L1 data cache that is used for most load and store operations.

Memory Subsystem

Figure 1 also shows the memory subsystem. This includes the L2 cache and the system bus. The L2 cache stores both instructions and data that cannot fit in the Execution Trace Cache and the L1 data cache. The external system bus is connected to the backside of the second-level cache and is used to access main memory when the L2 cache has a cache miss, and to access the system I/O resources.

CLOCK RATES

Processor microarchitectures can be pipelined to different degrees. The degree of pipelining is a microarchitectural decision. The final frequency of a specific processor pipeline on a given silicon process technology depends heavily on how deeply the processor is pipelined. When designing a new processor, a key design decision is the target design frequency of operation. The frequency target determines how many gates of logic can be included per pipeline stage in the design. This then helps determine how many pipeline stages there are in the machine.

There are tradeoffs when designing for higher clock rates. Higher clock rates need deeper pipelines so the efficiency at the same clock rate goes down. Deeper pipelines make many things take more clock cycles, such as mispredicted branches and cache misses, but usually more than make up for the lower per-clock efficiency by allowing the design to run at a much higher clock rate. For example, a 50% increase in frequency might buy only a 30% increase in net performance, but this frequency increase still provides a significant overall performance increase. High-frequency design also depends heavily on circuit design techniques, design methodology, design tools, silicon process technology, power and thermal constraints, etc. At higher frequencies, clock skew and jitter and latch delay become a much bigger percentage of the clock cycle, reducing the percentage of the clock cycle usable by actual logic. The deeper pipelines make the machine more complicated and require it to have deeper buffering to cover the longer pipelines.

Historical Trend of Processor Frequencies

Figure 2 shows the relative clock frequency of Intel's last six processor cores. The vertical axis shows the relative clock frequency, and the horizontal axis shows the various processors relative to each other.

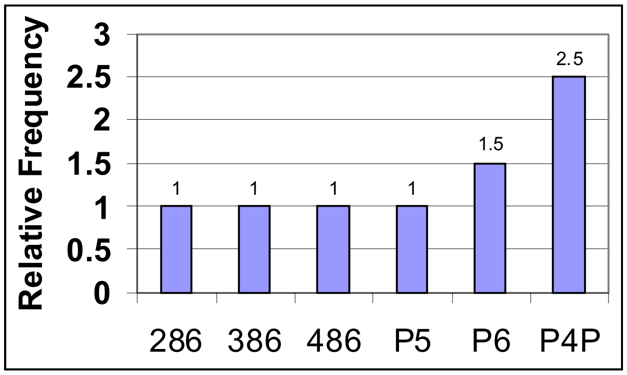


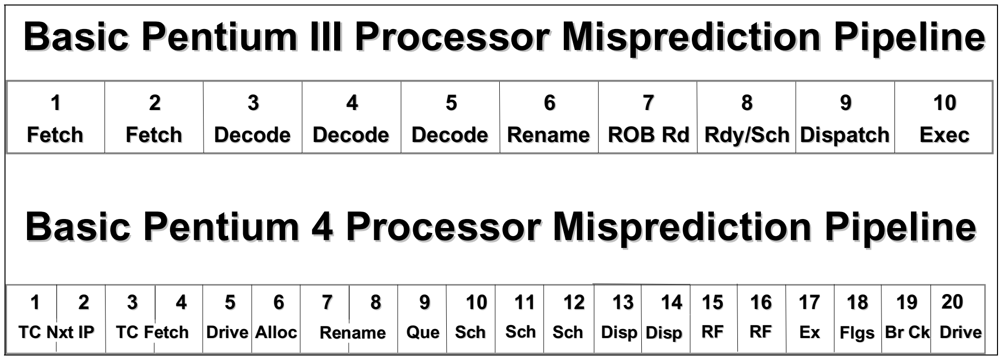
Figure 2: Relative frequencies of Intel's processors

Figure 2 shows that the 286, Intel386™, Intel486™ and Pentium® (P5) processors had similar pipeline depths– they would run at similar clock rates if they were all implemented on the same silicon process technology. They all have a similar number of gates of logic per clock cycle. The P6 microarchitecture lengthened the processor pipelines, allowing fewer gates of logic per pipeline stage, which delivered significantly higher frequency and performance. The P6 microarchitecture approximately doubled the number of pipeline stages compared to the earlier processors and was able to achieve about a 1.5 times higher frequency on the same process technology.

The NetBurst microarchitecture was designed to have an even deeper pipeline (about two times the P6 microarchitecture) with even fewer gates of logic per clock cycle to allow an industry-leading clock rate. Compared to the P6 family of processors, the Pentium 4 processor was designed with a greater than 1.6 times higher frequency target for its main clock rate, on the same process technology. This allows it to operate at a much higher frequency than the P6 family of processors on the same silicon process technology. At its introduction in November 2000, the Pentium 4 processor was at 1.5 times the frequency of the Pentium III processor. Over time this frequency delta will increase as the Pentium 4 processor design matures.

Different parts of the Pentium 4 processor run at different clock frequencies. The frequency of each section of logic is set to be appropriate for the performance it needs to achieve. The highest frequency section (fast clock) was set equal to the speed of the critical ALU-bypass execution loop that is used for most instructions in integer programs. Most other parts of the chip run at half of the 3GHz fast clock since this makes these parts much easier to design. A few sections of the chip run at a quarter of this fast-clock frequency making them also easier to design. The bus logic runs at 100MHz, to match the system bus needs.

As an example of the pipelining differences, Figure 3 shows a key pipeline in both the P6 and the Pentium 4 processors: the mispredicted branch pipeline. This pipeline covers the cycles it takes a processor to recover from a branch that went a different direction than the early fetch hardware predicted at the beginning of the machine pipeline. As shown, the Pentium 4 processor has a 20-stage misprediction pipeline while the P6 microarchitecture has a 10-stage misprediction pipeline. By dividing the pipeline into smaller pieces, doing less work during each pipeline stage (fewer gates of logic), the clock rate can be a lot higher.



Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Figure 3: Misprediction Pipeline

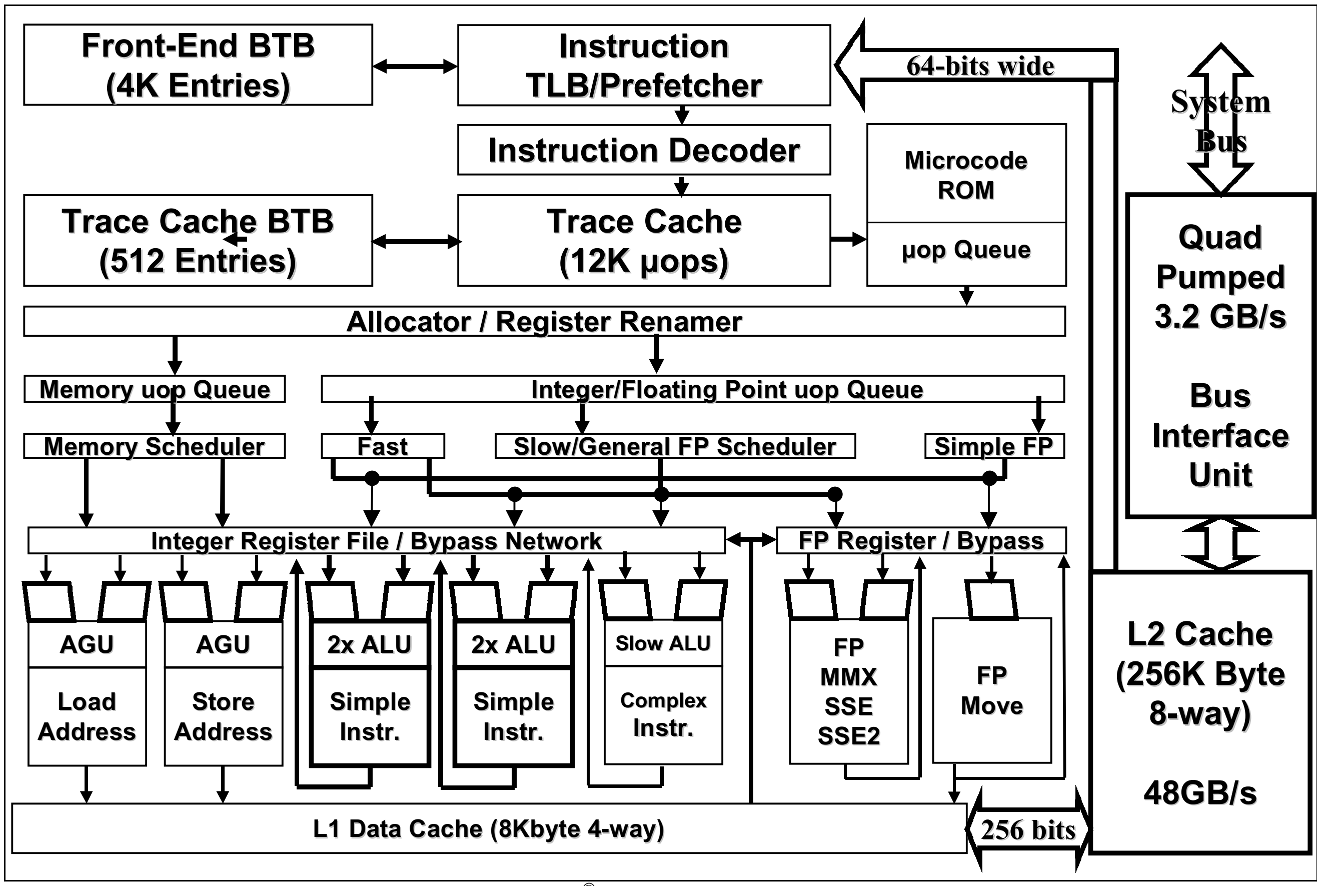


Figure 4: Pentium® 4 processor microarchitecture

NETBURST MICROARCHITECTURE

Figure 4 shows a more detailed block diagram of the NetBurst microarchitecture of the Pentium 4 processor. The top-left portion of the diagram shows the front end of the machine. The middle of the diagram illustrates the out-of-order buffering logic, and the bottom of the diagram shows the integer and floating-point execution units and the L1 data cache. On the right of the diagram is the memory subsystem.

Front End

The front end of the Pentium 4 processor consists of several units as shown in the upper part of Figure 4. It has the Instruction TLB (ITLB), the front-end branch predictor (labeled here Front-End BTB), the IA-32 Instruction Decoder, the Trace Cache, and the Microcode ROM.

Trace Cache

The Trace Cache is the primary or Level 1 (L1) instruction cache of the Pentium 4 processor and delivers up to three uops per clock to the out-of-order execution logic. Most instructions in a program are fetched and executed from the Trace Cache. Only when there is a Trace Cache miss does the NetBurst microarchitecture fetch and decode instructions from the Level 2 (L2) cache. This occurs about as often as previous processors miss their L1 instruction cache. The Trace Cache has a capacity to hold up to 12K uops. It has a similar hit rate to an 8K to 16K byte conventional instruction cache.

IA-32 instructions are cumbersome to decode. The instructions have a variable number of bytes and have many different options. The instruction decoding logic needs to sort this all out and convert these complex instructions into simple uops that the machine knows how to execute. This decoding is especially difficult when trying to decode several IA-32 instructions each clock cycle when running at the high clock frequency of the Pentium 4 processor. A high-bandwidth IA-32 decoder, that is capable of decoding several instructions per clock cycle, takes several pipeline stages to do its work. When a branch is mispredicted, the recovery time is much shorter if the machine does not have to re-decode the IA-32 instructions needed to resume execution at the corrected branch target location. By caching the uops of the previously decoded instructions in the Trace Cache, the NetBurst microarchitecture bypasses the instruction decoder most of the time thereby reducing misprediction latency and allowing the decoder to be simplified: it only needs to decode one IA-32 instruction per clock cycle.

The Execution Trace Cache takes the already-decoded uops from the IA-32 Instruction Decoder and assembles or builds them into program-ordered sequences of uops called traces. It packs the uops into groups of six uops per trace line. There can be many trace lines in a single trace. These traces consist of uops running sequentially down the predicted path of the IA-32 program execution. This allows the target of a branch to be included in the same trace cache line as the branch itself even if the branch and its target instructions are thousands of bytes apart in the program.

Conventional instruction caches typically provide instructions up to and including a taken branch instruction but none after it during that clock cycle. If the branch is the first instruction in a cache line, only the single branch instruction is delivered that clock cycle. Conventional instruction caches also often add a clock delay getting to the target of the taken branch, due to delays getting through the branch predictor and then accessing the new location in the instruction cache. The Trace Cache avoids both aspects of this instruction delivery delay for programs that fit well in the Trace Cache.

The Trace Cache has its own branch predictor that directs where instruction fetching needs to go next in the Trace Cache. This Trace Cache predictor (labeled Trace BTB in Figure 4) is smaller than the front-end predictor, since its main purpose is to predict the branches in the subset of the program that is currently in the Trace Cache. The branch prediction logic includes a 16-entry return address stack to efficiently predict return addresses, because often the same procedure is called from several different call sites. The Trace-Cache BTB, together with the front-end BTB, use a highly advanced branch prediction algorithm that reduces the branch misprediction rate by about 1/3 compared to the predictor in the P6 microarchitecture.

Microcode ROM

Near the Trace Cache is the microcode ROM. This ROM is used for complex IA-32 instructions, such as string move, and for fault and interrupt handling. When a complex instruction is encountered, the Trace Cache jumps into the microcode ROM which then issues the uops needed to complete the operation. After the microcode ROM finishes sequencing uops for the current IA-32 instruction, the front end of the machine resumes fetching uops from the Trace Cache.

The uops that come from the Trace Cache and the microcode ROM are buffered in a simple, in-order uop queue that helps smooth the flow of uops going to the out-of-order execution engine.

ITLB and Front-End BTB

The IA-32 Instruction TLB and front-end BTB, shown at the top of Figure 4, steer the front end when the machine misses the Trace Cache. The ITLB translates the linear instruction pointer addresses given to it into physical addresses needed to access the L2 cache. The ITLB also performs page-level protection checking.

Hardware instruction prefetching logic associated with the front-end BTB fetches IA-32 instruction bytes from the L2 cache that are predicted to be executed next. The fetch logic attempts to keep the instruction decoder fed with the next IA-32 instructions the program needs to execute. This instruction prefetcher is guided by the branch prediction logic (branch history table and branch target buffer listed here as the front-end BTB) to know what to fetch next. Branch prediction allows the processor to begin fetching and executing instructions long before the previous branch outcomes are certain. The front-end branch predictor is quite large–4K branch target entries–to capture most of the branch history information for the program. If a branch is not found in the BTB, the branch prediction hardware statically predicts the outcome of the branch based on the direction of the branch displacement (forward or backward). Backward branches are assumed to be taken and forward branches are assumed to not be taken.

IA-32 Instruction Decoder

The instruction decoder receives IA-32 instruction bytes from the L2 cache 64-bits at a time and decodes them into primitives, called uops, that the machine knows how to execute. This single instruction decoder can decode at a maximum rate of one IA-32 instruction per clock cycle. Many IA-32 instructions are converted into a single uop, and others need several uops to complete the full operation. If more than four uops are needed to complete an IA-32 instruction, the decoder sends the machine into the microcode ROM to do the instruction. Most instructions do not need to jump to the microcode ROM to complete. An example of a many-uop instruction is string move, which could have thousands of uops.

Out-of-Order Execution Logic

The out-of-order execution engine consists of the allocation, renaming, and scheduling functions. This part of the machine re-orders instructions to allow them to execute as quickly as their input operands are ready.

The processor attempts to find as many instructions as possible to execute each clock cycle. The out-of-order execution engine will execute as many ready instructions as possible each clock cycle, even if they are not in the original program order. By looking at a larger number of instructions from the program at once, the out-of-order execution engine can usually find more ready-to-execute, independent instructions to begin. The NetBurst microarchitecture has much deeper buffering than the P6 microarchitecture to allow this. It can have up to 126 instructions in flight at a time and have up to 48 loads and 24 stores allocated in the machine at a time.

The Allocator

The out-of-order execution engine has several buffers to perform its re-ordering, tracking, and sequencing operations. The Allocator logic allocates many of the key machine buffers needed by each uop to execute. If a needed resource, such as a register file entry, is unavailable for one of the three uops coming to the Allocator this clock cycle, the Allocator will stall this part of the machine. When the resources become available the Allocator assigns them to the requesting uops and allows these satisfied uops to flow down the pipeline to be executed. The Allocator allocates a Reorder Buffer (ROB) entry, which tracks the completion status of one of the 126 uops that could be in flight simultaneously in the machine. The Allocator also allocates one of the 128 integer or floating-point register entries for the result data value of the uop, and possibly a load or store buffer used to track one of the 48 loads or 24 stores in the machine pipeline. In addition, the Allocator allocates an entry in one of the two uop queues in front of the instruction schedulers.

Register Renaming

The register renaming logic renames the logical IA-32 registers such as EAX onto the processors 128-entry physical register file. This allows the small, 8-entry, architecturally defined IA-32 register file to be dynamically expanded to use the 128 physical registers in the Pentium 4 processor. This renaming process removes false conflicts caused by multiple instructions creating their simultaneous but unique versions of a register such as EAX. There could be dozens of unique instances of EAX in the machine pipeline at one time. The renaming logic remembers the most current version of each register, such as EAX, in the Register Alias Table (RAT) so that a new instruction coming down the pipeline can know where to get the correct current instance of each of its input operand registers.

As shown in Figure 5 the NetBurst microarchitecture allocates and renames the registers somewhat differently than the P6 microarchitecture. On the left of Figure 5, the P6 scheme is shown. It allocates the data result registers and the ROB entries as a single, wide entity with a data and a status field. The ROB data field is used to store the data result value of the uop, and the ROB status field is used to track the status of the uop as it is executing in the machine. These ROB entries are allocated and deallocated sequentially and are pointed to by a sequence number that indicates the relative age of these entries. Upon retirement, the result data is physically copied from the ROB data result field into the separate Retirement Register File (RRF). The RAT points to the current version of each of the architectural registers such as EAX. This current register could be in the ROB or in the RRF.

The NetBurst microarchitecture allocation scheme is shown on the right of Figure 5. It allocates the ROB entries and the result data Register File (RF) entries separately.

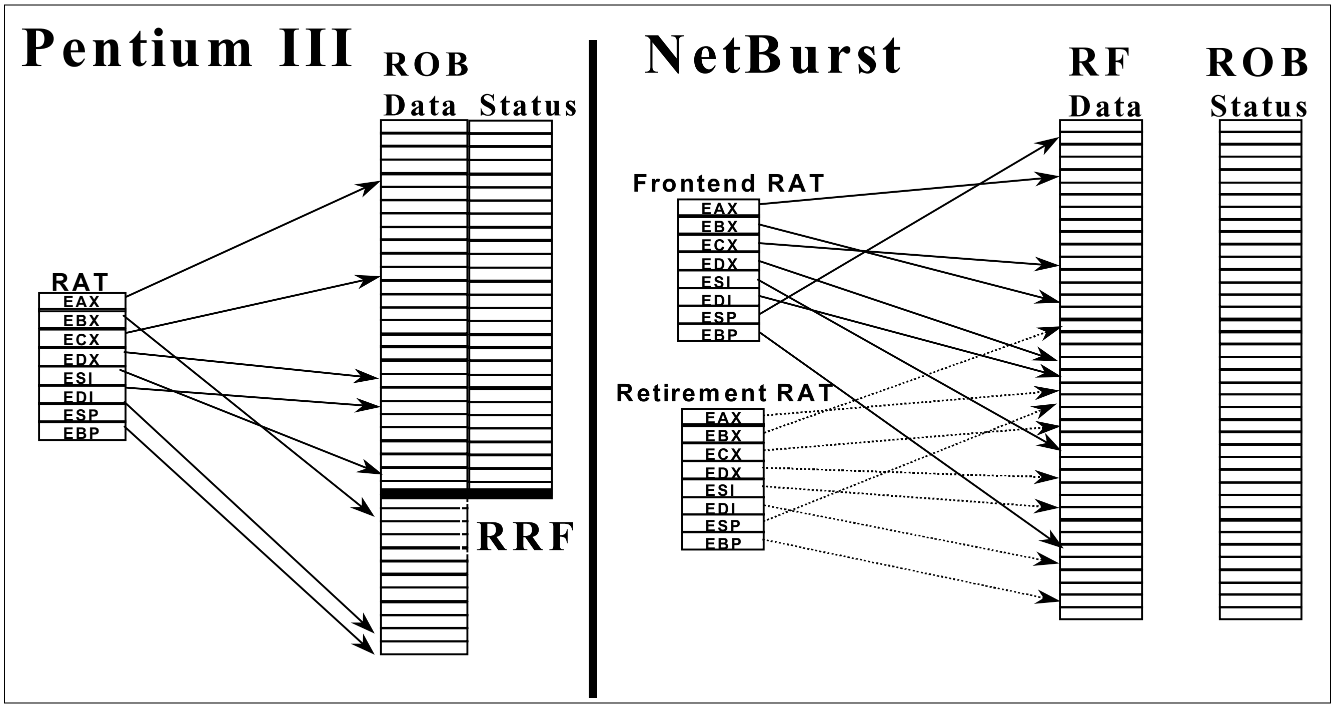


Figure 5: Pentium® III vs. Pentium® 4 processor register allocation

The ROB entries, which track uop status, consist only of the status field and are allocated and deallocated sequentially. A sequence number assigned to each uop indicates its relative age. The sequence number points to the uop's entry in the ROB array, which is similar to the P6 microarchitecture. The Register File entry is allocated from a list of available registers in the 128-entry RF–not sequentially like the ROB entries. Upon retirement, no result data values are actually moved from one physical structure to another.

Uop Scheduling

The uop schedulers determine when a uop is ready to execute by tracking its input register operands. This is the heart of the out-of-order execution engine. The uop schedulers are what allow the instructions to be reordered to execute as soon as they are ready, while still maintaining the correct dependencies from the original program. The NetBurst microarchitecture has two sets of structures to aid in uop scheduling: the uop queues and the actual uop schedulers.

There are two uop queues–one for memory operations (loads and stores) and one for non-memory operations. Each of these queues stores the uops in strict FIFO (first-in, first-out) order with respect to the uops in its own queue, but each queue is allowed to be read out-of-order with respect to the other queue. This allows the dynamic out-of-order scheduling window to be larger than just having the uop schedulers do all the reordering work.

There are several individual uop schedulers that are used to schedule different types of uops for the various execution units on the Pentium 4 processor as shown in Figure 6. These schedulers determine when uops are ready to execute based on the readiness of their dependent input register operand sources and the availability of the execution resources the uops need to complete their operation.

These schedulers are tied to four different dispatch ports. There are two execution unit dispatch ports labeled port 0 and port 1 in Figure 6. These ports are fast: they can dispatch up to two operations each main processor clock cycle. Multiple schedulers share each of these two dispatch ports. The fast ALU schedulers can schedule on each half of the main clock cycle while the other schedulers can only schedule once per main processor clock cycle. They arbitrate for the dispatch port when multiple schedulers have ready operations at once. There is also a load and a store dispatch port that can dispatch a ready load and store each clock cycle. Collectively, these uop dispatch ports can dispatch up to six uops each main clock cycle. This dispatch bandwidth exceeds the front-end and retirement bandwidth, of three uops per clock, to allow for peak bursts of greater than 3 uops per clock and to allow higher flexibility in issuing uops to different dispatch ports. Figure 6 also shows the types of operations that can be dispatched to each port each clock cycle.

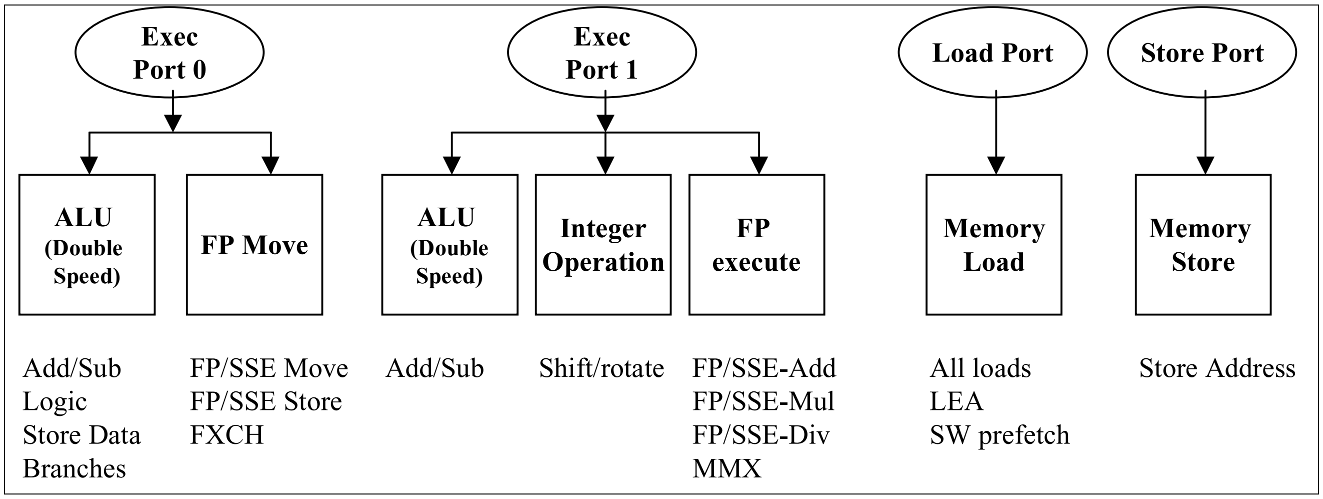


Figure 6: Dispatch ports in the Pentium® 4 processor

Integer and Floating-Point Execution Units

The execution units are where the instructions are actually executed. The execution units are designed to optimize overall performance by handling the most common cases as fast as possible. There are several different execution units in the NetBurst microarchitecture. The units used to execute integer operations include the low-latency integer ALUs, the complex integer instruction unit, the load and store address generation units, and the L1 data cache.

Floating-Point (x87), MMX, SSE (Streaming SIMD Extension), and SSE2 (Streaming SIMD Extension 2) operations are executed by the two floating-point execution blocks. MMX instructions are 64-bit packed integer SIMD operations that operate on 8, 16, or 32-bit operands. The SSE instructions are 128-bit packed IEEE single-precision floating-point operations. The Pentium 4 processor adds new forms of 128-bit SIMD instructions called SSE2. The SSE2 instructions support 128-bit packed IEEE double-precision SIMD floating-point operations and 128-bit packed integer SIMD operations. The packed integer operations support 8, 16, 32, and 64-bit operands. See IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture [3] for more detail on these SIMD operations.

The Integer and floating-point register files sit between the schedulers and the execution units. There is a separate 128-entry register file for both the integer and the floating-point/SSE operations. Each register file also has a multi-clock bypass network that bypasses or forwards just-completed results, which have not yet been written into the register file, to the new dependent uops. This multi-clock bypass network is needed because of the very high frequency of the design.

Low Latency Integer ALU

The Pentium 4 processor execution units are designed to optimize overall performance by handling the most common cases as fast as possible. The Pentium 4 processor can do fully dependent ALU operations at twice the main clock rate. The ALU-bypass loop is a key closed loop in the processor pipeline. Approximately 60-70% of all uops in typical integer programs use this key integer ALU loop. Executing these operations at ½ the latency of the main clock helps speed up program execution for most programs. Doing the ALU operations in one half a clock cycle does not buy a 2x performance increase, but it does improve the performance for most integer applications.

This high-speed ALU core is kept as small as possible to minimize the metal length and loading. Only the essential hardware necessary to perform the frequent ALU operations is included in this high-speed ALU execution loop. Functions that are not used very frequently, for most integer programs, are not put in this key low-latency ALU loop but are put elsewhere. Some examples of integer execution hardware put elsewhere are the multiplier, shifts, flag logic, and branch processing.

The processor does ALU operations with an effective latency of one-half of a clock cycle. It does this operation in a sequence of three fast clock cycles (the fast clock runs at 2x the main clock rate) as shown in Figure 7. In the first fast clock cycle, the low order 16-bits are computed and are immediately available to feed the low 16-bits of a dependent operation the very next fast clock cycle. The high-order 16 bits are processed in the next fast cycle, using the carry out just generated by the low 16-bit operation. This upper 16-bit result will be available to the next dependent operation exactly when needed. This is called a staggered add. The ALU flags are processed in the third fast cycle. This staggered add means that only a 16-bit adder and its input muxes need to be completed in a fast clock cycle. The low order 16 bits are needed at one time in order to begin the access of the L1 data cache when used as an address input.

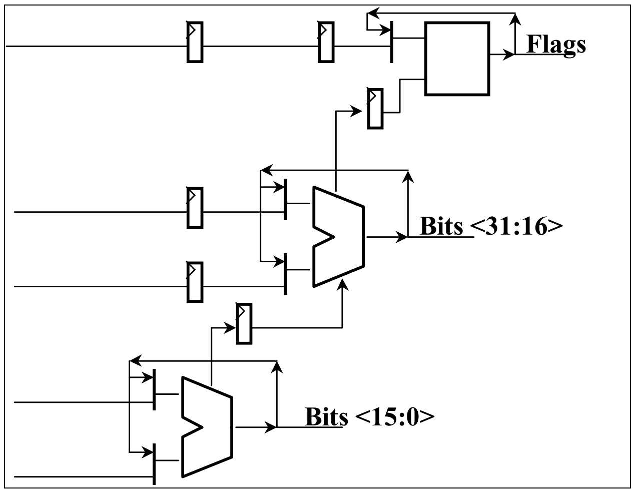


Figure 7: Staggered ALU add

Complex Integer Operations

The simple, very frequent ALU operations go to the high-speed integer ALU execution units described above. Integer operations that are more complex go to separate hardware for completion. Most integer shift or rotate operations go to the complex integer dispatch port. These shift operations have a latency of four clocks. Integer multiply and divide operations also have a long latency. Typical forms of multiply and divide have a latency of about 14 and 60 clocks, respectively.

Low Latency Level 1 (L1) Data Cache

The Level 1 (LI) data cache is an 8K-byte cache that is used for both integer and floating-point/SSE loads and stores. It is organized as a 4-way set-associative cache that has 64 bytes per cache line. It is a write-through cache, which means that writes to it are always copied into the L2 cache. It can do one load and one store per clock cycle.

The latency of load operations is a key aspect of processor performance. This is especially true for IA-32 programs that have a lot of loads and stores because of the limited number of registers in the instruction set. The NetBurst microarchitecture optimizes for the lowest overall load-access latency with a small, very low latency 8K byte cache backed up by a large, high-bandwidth second-level cache with medium latency. For most IA-32 programs this configuration of a small, but very low latency, L1 data cache followed by a large medium-latency L2 cache gives lower net load-access latency and therefore higher performance than a bigger, slower L1 cache. The L1 data cache operates with a 2-clock load-use latency for integer loads and a 6-clock load-use latency for floating-point/SSE loads.

This 2-clock load latency is hard to achieve with the very high clock rates of the Pentium 4 processor. This cache uses new access algorithms to enable this very low load-access latency. The new algorithm leverages the fact that almost all accesses hit the first-level data cache and the data TLB (DTLB).

At this high frequency and with this deep machine pipeline, the distance in clocks, from the load scheduler to execution, is longer than the load execution latency itself. The uop schedulers dispatch dependent operations before the parent load has finished executing. In most cases, the scheduler assumes that the load will hit the L1 data cache. If the load misses the L1 data cache, there will be dependent operations in flight in the pipeline. These dependent operations that have left the scheduler will get temporarily incorrect data. This is a form of data speculation. Using a mechanism known as *replay*, logic tracks and re-executes instructions that use incorrect data. Only the dependent operations are replayed: the independent ones are allowed to complete.

There can be up to four outstanding load misses from the L1 data cache pending at any one time in the memory subsystem.

Store-to-Load Forwarding

In an out-of-order-execution processor, stores are not allowed to be committed to permanent machine state (the L1 data cache, etc.) until after the store has retired. Waiting until retirement means that all other preceding operations have completely finished. All faults, interrupts, mispredicted branches, etc. must have been signaled beforehand to make sure this store is safe to perform. With the very deep pipeline of the Pentium 4 processor it takes many clock cycles for a store to make it to retirement. Also, stores that are at retirement often have to wait for previous stores to complete their update of the data cache. This machine can have up to 24 stores in the pipeline at a time. Sometimes many of them have retired but have not yet committed their state into the L1 data cache. Other stores may have completed, but have not yet retired, so their results are also not yet in the L1 data cache. Often loads must use the result of one of these pending stores, especially for IA-32 programs, due to the limited number of registers available. To enable this use of pending stores, modern out-of-order execution processors have a pending store buffer that allows loads to use the pending store results before the stores have been written into the L1 data cache. This process is called store-to-load forwarding.

To make this store-to-load-forwarding process efficient, this pending store buffer is optimized to allow efficient and quick forwarding of data to dependent loads from the pending stores. The Pentium 4 processor has a 24-entry store-forwarding buffer to match the number of stores that can be in flight at once. This forwarding is allowed if a load hits the same address as a proceeding, completed, pending store that is still in the store-forwarding buffer. The load must also be the same size or smaller than the pending store and have the same beginning physical address as the store, for the forwarding to take place. This is by far the most common forwarding case. If the bytes requested by a load only partially overlap a pending store or need to have some bytes come simultaneously from more than one pending store, this store-to-load forwarding is not allowed. The load must get its data from the cache and cannot complete until the store has committed its state to the cache.

This disallowed store-to-load forwarding case can be quite costly, in terms of performance loss, if it happens very often. When it occurs, it tends to happen on older P5-core optimized applications that have not been optimized for modern, out-of-order execution microarchitectures. The newer versions of the IA-32 compilers remove most or all of these bad store-to-load forwarding cases but they are still found in many old legacy P5 optimized applications and benchmarks. This bad store-forwarding case is a big performance issue for P6-based processors and other modern processors, but due to the even deeper pipeline of the Pentium 4 processor, these cases are even more costly in performance.

FP/SSE Execution Units

The Floating-Point (FP) execution cluster of the Pentium 4 processor is where the floating-point, MMX, SSE, and SSE2 instructions are executed. These instructions typically have operands from 64 to 128 bits in width. The FP/SSE register file has 128 entries and each register is 128 bits wide. This execution cluster has two 128-bit execution ports that can each begin a new operation every clock cycle. One execution port is for 128-bit general execution and one is for 128-bit register-to-register moves and memory stores. The FP/SSE engine can also complete a full 128-bit load each clock cycle.

Early in the development cycle of the Pentium 4 processor, we had two full FP/SSE execution units, but this cost a lot of hardware and did not buy very much performance for most FP/SSE applications. Instead, we optimized the cost/performance tradeoff with a simple second port that does FP/SSE moves and FP/SSE store data primitives. This tradeoff was shown to buy most of the performance of a second full-featured port with much less die size and power cost.

Many FP/multi-media applications have a fairly balanced set of multiplies and adds. The machine can usually keep busy interleaving a multiply and an add every two clock cycles at much less cost than fully pipelining all the FP/SSE execution hardware. In the Pentium 4 processor, the FP adder can execute one Extended-Precision (EP) addition, one Double-Precision (DP) addition, or two Single-Precision (SP) additions every clock cycle. This allows it to complete a 128-bit SSE/SSE2 packed SP or DP add uop every two clock cycles. The FP multiplier can execute either one EP multiply every two clocks, or it can execute one DP multiply or two SP multiplies every clock. This allows it to complete a 128-bit IEEE SSE/SSE2 packed SP or DP multiply uop every two clock cycles giving a peak 6 GFLOPS for single precision or 3 GFLOPS for double precision floating-point at 1.5GHz.

Many multi-media applications interleave adds, multiplies, and pack/unpack/shuffle operations. For integer SIMD operations, which are the 64-bit wide MMX or 128-bit wide SSE2 instructions, there are three execution units that can run in parallel. The SIMD integer ALU execution hardware can process 64 SIMD integer bits per clock cycle. This allows the unit to do a new 128-bit SSE2 packed integer add uop every two clock cycles. A separate shuffle/unpack execution unit can also process 64 SIMD integer bits per clock cycle allowing it to do a full 128-bit shuffle/unpack uop operation each two clock cycles. MMX/SSE2 SIMD integer multiply instructions use the FP multiply hardware mentioned above to also do a 128-bit packed integer multiply uop every two clock cycles.

The FP divider executes all divide, square root, and remainder uops. It is based on a double-pumped SRT radix-2 algorithm, producing two bits of quotient (or square root) every clock cycle.

Achieving significantly higher floating-point and multi-media performance requires much more than just fast execution units. It requires a balanced set of capabilities that work together. These programs often have many long latency operations in their inner loops. The very deep buffering of the Pentium 4 processor (126 uops and 48 loads in flight) allows the machine to examine a large section of the program at once. The out-of-order-execution hardware often unrolls the inner execution loop of these programs numerous times in its execution window. This dynamic unrolling allows the Pentium 4 processor to overlap the long-latency FP/SSE and memory instructions by finding many independent instructions to work on simultaneously. This deep window buys a lot more performance for most FP/multi-media applications than more execution units would.

FP/multi-media applications usually need a very high bandwidth memory subsystem. Sometimes FP and multi-media applications do not fit well in the L1 data cache but do fit in the L2 cache. To optimize these applications the Pentium 4 processor has a high bandwidth path from the L2 data cache to the L1 data. Some FP/multi-media applications stream data from memory–no practical cache size will hold the data. They need a high bandwidth path to main memory to perform well. The long 128-byte L2 cache lines together with the hardware prefetcher described below help to prefetch the data that the application will soon need, effectively hiding the long memory latency. The high bandwidth system bus of the Pentium 4 processor allows this prefetching to help keep the execution engine well fed with streaming data.

Memory Subsystem

The Pentium 4 processor has a highly capable memory subsystem to enable the new, emerging, high-bandwidth stream-oriented applications such as 3D, video, and content creation. The memory subsystem includes the Level 2 (L2) cache and the system bus. The L2 cache stores data that cannot fit in the Level 1 (L1) caches. The external system bus is used to access main memory when the L2 cache has a cache miss and also to access the system I/O devices.

Level 2 Instruction and Data Cache

The L2 cache is a 256K-byte cache that holds both instructions that miss the Trace Cache and data that miss the L1 data cache. The L2 cache is organized as an 8-way set-associative cache with 128 bytes per cache line. These 128-byte cache lines consist of two 64-byte sectors. A miss in the L2 cache typically initiates two 64-byte access requests to the system bus to fill both halves of the cache line. The L2 cache is a write-back cache that allocates new cache lines on load or store misses. It has a net load-use access latency of seven clock cycles. A new cache operation can begin every two processor clock cycles for a peak bandwidth of 48Gbytes per second, when running at 1.5GHz.

Associated with the L2 cache is a hardware prefetcher that monitors data access patterns and prefetches data automatically into the L2 cache. It attempts to stay 256 bytes ahead of the current data access locations. This prefetcher remembers the history of cache misses to detect concurrent, independent streams of data that it tries to prefetch ahead of use in the program. The prefetcher also tries to minimize prefetching unwanted data that can cause over utilization of the memory system and delay the real accesses the program needs.

400MHz System Bus

The Pentium 4 processor has a system bus with 3.2 Gbytes per second of bandwidth. This high bandwidth is a key enabler for applications that stream data from memory. This bandwidth is achieved with a 64-bit wide bus capable of transferring data at a rate of 400MHz. It uses a source-synchronous protocol that quad-pumps the 100MHz bus to give 400 million data transfers per second. It has a split-transaction, deeply pipelined protocol to allow the memory subsystem to overlap many simultaneous requests to actually deliver high memory bandwidths in a real system. The bus protocol has a 64-byte access length.

PERFORMANCE

The Pentium 4 processor delivers the highest SPECint_base performance of any processor in the world. It also delivers world-class SPECfp2000 performance. These are industry standard benchmarks that evaluate general integer and floating-point application performance.

Figure 8 shows the performance comparison of a Pentium 4 processor at 1.5GHz compared to a Pentium III processor at 1GHz for various applications. The integer applications are in the 15-20% performance gain while the FP and multi-media applications are in the 30-70% performance advantage range. For FSPEC 2000 the new SSE/SSE2 instructions buy about 5% performance gain compared to an x87-only version. As the compiler improves over time the gain from these new instructions will increase. Also, as the relative frequency of the Pentium 4 processor increases over time (as its design matures), all these performance deltas will increase.

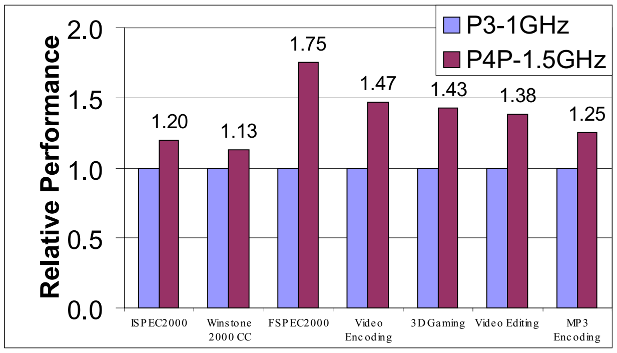


Figure 8: Performance comparison

For a more complete performance brief covering many application performance areas on the Pentium 4 processor, go to http://www.intel.com/procs/perf/pentium4/.

CONCLUSION

The Pentium 4 processor is a new, state-of-the-art processor microarchitecture and design. It is the beginning of a new family of processors that utilize the new Intel NetBurst microarchitecture. Its deeply pipelined design delivers world-leading frequencies and performance. It uses many novel microarchitectural ideas including a Trace Cache, double-clocked ALU, new low-latency L1 data cache algorithms, and a new high bandwidth system bus. It delivers world-class performance in the areas where added performance makes a difference including media rich environments (video, sound, and speech), 3D applications, workstation applications, and content creation.

ACKNOWLEDGMENTS

The authors thank all the architects, designers, and validators who contributed to making this processor into a real product.

REFERENCES

1. D. Sager, G. Hinton, M. Upton, T. Chappell, T. Fletcher, S. Samaan, and R. Murray, "A 0.18um CMOS IA32 Microprocessor with a 4GHz Integer Execution Unit," International Solid State Circuits Conference, Feb 2001.

2. Doug Carmean, "Inside the High-Performance Intel® Pentium® 4 Processor Micro-architecture" Intel Developer Forum, Fall 2000 at ftp://download.intel.com/design/idf/fall2000/presentations/pda/pda_s01_cd.pdf

3. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture at http://developer.intel.com/design/pentium4/manuals/245470.htm.

4. Intel® Pentium® 4 Processor Optimization Reference Manual at http://developer.intel.com/design/pentium4/manuals/248966.htm.

AUTHORS' BIOGRAPHIES

Glenn Hinton is an Intel Fellow and Director of IA-32 Microarchitecture Development in the Intel Architecture Group. Hinton joined Intel in 1983. He was one of three senior architects in 1990 responsible for the P6 processor microarchitecture, which became the Pentium® Pro, Pentium® II, Pentium® III, and Celeron™ processors. He was responsible for the microarchitecture development of the Pentium® 4 processor. Hinton received a master's degree in Electrical Engineering from Brigham Young University in 1983. His e-mail address is glenn.hinton@intel.com.

Dave Sager is a Principal Engineer/Architect in Intel's Desktop Platforms Group, and is one of the overall architects of the Intel® Pentium 4 processor. He joined Intel in 1995. Dave also worked for 17 years at Digital Equipment Corporation in their processor research labs. He graduated from Princeton University with a Ph.D. in Physics in 1973. His e-mail address is dave.sager@intel.com.

Michael Upton is a Principal Engineer/Architect in Intel's Desktop Platforms Group, and is one of the architects of the Intel® Pentium 4 processor. He completed B.S. and M.S. degrees in Electrical Engineering from the University of Washington in 1985 and 1990. After a number of years in IC design and CAD tool development, he entered the University of Michigan to study computer architecture. Upon completion of his Ph.D degree in 1994, he joined Intel to work on the Pentium® Pro and Pentium 4 processors. His e-mail address is mike.upton@intel.com.

Darrell Boggs is a Principal Engineer/Architect with Intel Corporation and has been working as a microarchitect for nearly 10 years. He graduated from Brigham Young University with a M.S. in Electrical Engineering. Darrell played a key role on the Pentium® Pro Processor design, and was one of the key architects of the Pentium 4 Processor. Darrell holds many patents in the areas of register renaming; instruction decoding; events and state recovery mechanisms. His e-mail address is darrell.boggs@intel.com.

Douglas M. Carmean is a Principal Engineer/Architect with Intel's Desktop Products Group in Oregon. Doug was one of the key architects, responsible for definition of the Intel Pentium® 4 processor. He has been with Intel for 12 years, working on IA-32 processors from the 80486 to the Intel Pentium 4 processor and beyond. Prior to joining Intel, Doug worked at ROSS Technology, Sun Microsystems, Cypress Semiconductor and Lattice Semiconductor. Doug enjoys fast cars and scary, Italian motorcycles. His e-mail address is douglas.m.carmean@intel.com.

Patrice Roussel graduated from the University of Rennes in 1980 and L'Ecole Superieure d'Electricite in 1982 with a M.S. degree in signal processing and VLSI design. Upon graduation, he worked at Cimatel, an Intel/Matra Harris joint design center. He moved to the USA in 1988 to join Intel in Arizona and worked on the 960CA chip. In late 1991, he moved to Intel in Oregon to work on the P6 processors. Since 1995, he has been the floating-point architect of the Pentium® 4 processor. His e-mail address is patrice.roussel@intel.com.

Validating The Intel® Pentium® 4 Processor

Bob Bentley, Desktop Platforms Group, Intel Corp.
Rand Gray, Desktop Platforms Group, Intel Corp.

Index words: microprocessor, validation, bugs, verification

ABSTRACT

Developing a new leading-edge Intel® Architecture microprocessor is an immensely complicated undertaking. The microarchitecture of the Pentium® 4 processor is significantly more complex than any previous Intel Architecture microprocessor, so the challenge of validating the logical correctness of the design in a timely fashion was indeed a daunting one. In order to meet this challenge, we applied a number of innovative tools and methodologies, which enabled us to keep functional validation off the critical path to tapeout while meeting our goal of ensuring that first silicon was functional enough to boot operating systems and run applications. This in turn enabled the post-silicon validation teams to quickly "peel the onion", resulting in an elapsed time of only ten months from initial tapeout to production shipment qualification, an Intel record for a new IA-32 microarchitecture.

This paper describes how we went about the task of validating the Pentium 4 processor and what we found along the way. We hope that other microprocessor designers and validators will be able to benefit from our experience and insights. As Doug Clark has remarked "Finding a bug should be a cause for celebration. Each discovery is a small victory; each marks an incremental improvement in the design." [1]

INTRODUCTION

The Pentium 4 processor is Intel's most advanced IA-32 microprocessor, incorporating a host of new microarchitectural features including a 400MHz system bus, hyper pipelined technology, advanced dynamic execution, rapid execution engine, advanced transfer cache, execution trace cache, and Streaming Single Instruction, Multiple Data (SIMD) Extensions 2 (SSE2).

PRE-SILICON VALIDATION CHALLENGES AND ISSUES

The first thing that we had to do was build a validation team. Fortunately, we had a nucleus of people who had worked on the Pentium® Pro processor and who could do the initial planning for the Pentium 4 processor while at the same time working with the architects and designers who were refining the basic microarchitectural concepts. However, it was clear that a larger team would be needed, so we mounted an extensive recruitment campaign focused mostly on new college graduates. Not only did this take a large amount of effort from the original core team (at one stage we were spending an aggregate 25% of our total effort on recruiting!), but it also meant that we faced the monumental task of training these new team members. However, this investment paid off handsomely over the next few years as the team matured into a highly effective bug-finding machine that found more than 60% of all the logic bugs that were filed prior to tapeout. In doing so, they developed an in-depth knowledge of the Pentium 4 processor's NetBurst™ microarchitecture that has proved to be invaluable in post-silicon logic and speedpath debug and also in fault grade test writing.

For the most part, we applied the same or similar tools and methodologies that were used on the Pentium Pro processor to validate the Pentium 4 processor. However, we did develop new methodologies and tools in response to lessons learnt from previous projects and also to address some new challenges raised by the Pentium 4 processor design. In particular, the use of Formal Verification, Cluster Test Environments, and focused Power Reduction Validation was either new or a greatly extended form than that used on previous projects. These methodologies and tools are discussed in detail in later sections of this paper.

Pre-Silicon Validation Environment

Except for Formal Verification (FV), all pre-silicon validation was done using either a cluster-level or full-chip SRTL model running in the CSIM simulation environment developed by Intel Design Technology. We ran these simulation models on either interactive workstations or compute servers. Initially, these were legacy IBM RS/6000s* running AIX*, but over the course of the project we switched to systems based on the Pentium® III processor, running Linux*. Our computing pool grew to encompass several thousand systems by the end of the project, most of them compute servers. We used an internal tool called Netbatch to submit large numbers of batch simulations to these systems, which we were able to keep utilized at over 90% of their maximum 24/7 capacity. By tapeout we were averaging 5-6 billion cycles per week and had accumulated over 200 billion (to be precise, $2.384 * 10^{11}$) SRTL simulation cycles of all types.

Formal Verification

The Pentium 4 processor was the first project of its kind at Intel to apply FV on a large scale. We decided early in the project that the FV field had matured to the point where we could consider trying to use it as an integral part of the design verification process rather than only applying it retroactively, as had been done on previous products such as the Pentium Pro processor. However, it was clear from the start that we couldn't formally verify the entire design—that was (and still is) way beyond the state of the art for today's tools. We therefore decided to focus on the areas of the design where we believed FV could make a significant contribution; in particular, we focused on the floating-point execution units and the instruction decode logic. Because these areas had been sources of bugs in the past that escaped early detection, using FV allowed us to apply this technology to some real problems with real payback.

One of the major challenges for the FV team was to develop the tools and methodology needed to handle a large number of proofs in a highly dynamic environment. For the most part we took a model-checking approach to FV, using the Prover tool from Intel's Design Technology group to compare SRTL against separate specifications written in Formal Specification Language (FSL). By the time we taped out we had over 10,000 of these proofs in our proof database, each of which had to be maintained and regressed as the SRTL changed over the life of the project. Along the way, we found over 100 logic bugs—not a large number in the overall scheme of things, but 20 of them were "high-quality" bugs that we believe would not have been found by any of our other pre-silicon validation activities. Two of these bugs were classic floating-point data space problems:

1. The FADD instruction had a bug where, for a specific combination of source operands, the 72-bit FP adder was setting the carryout bit to 1 when there was no actual carryout.
2. The FMUL instruction had a bug where, when the rounding mode was set to "round up", the sticky bit was not set correctly for certain combinations of source operand mantissa values, specifically:

   $src1[67:0] := X*2^{(i+15)} + 1*2^{i}$

   $src2[67:0] := Y*2^{(j+15)} + 1*2^{j}$

   where $i+j = 54$, and $\{X,Y\}$ are any integers that fit in the 68-bit range.

Either of these bugs could easily have gone undetected[1], not just in the pre-silicon environment but also in post-silicon testing.

We put a lot of effort into making the regression of the FV proof database as push-button as possible, not only to simplify the task of running regressions against a moving SRTL target but because we viewed reuse as being one of the keys to proliferating the quality of the original design. This approach has had an immediate payoff: a regression of the database of 10,000 proofs on an early SRTL model of a proliferation of the Pentium 4 processor yielded a complex floating-point bug.

Cluster-Level Testing

One of the fundamental decisions that we took early in the Pentium 4 processor development program was to develop Cluster Test Environments (CTEs) and maintain them for the life of the project. There is a CTE for each of the six clusters into which the Pentium 4 processor design is logically subdivided (actually, microcode can be considered to be a seventh logical cluster, and it too has a test environment equivalent to the other CTEs). These CTEs are groupings of logically related units (e.g., all the execution units of the machine constitute one CTE) surrounded by code that emulates the interfaces to adjacent units outside of the cluster and provides an environment for creating and running tests and checking results.

These CTEs took a good deal of effort to develop and maintain, and were themselves a source of a significant number of bugs. However, they provided a number of key advantages:

First and foremost, they provided **controllability** that was otherwise lacking at the full-chip level. An out of order, speculative execution engine like the Pentium® Pro or Pentium 4 processor is inherently difficult to control at the instruction set architecture level. Assembly-language instructions (macroinstructions) are broken down by the machine into sequences of microinstructions that may be executed in any order (subject to data dependencies) relative to one another and to microinstructions from other preceding or following macroinstructions. Trying to produce precise microarchitectural behavior from macroinstruction sequences has been likened to pushing on a piece of string. This problem is particularly acute for the back end of the machine, the memory and bus clusters that lie beyond the out-of-order section of the microarchitecture pipeline. CTEs allowed us to provoke specific microarchitectural behavior on demand.

Second, CTEs allowed us to make significant strides in **early validation** of the Pentium 4 processor SRTL even

[1] We calculated that the probability of hitting the FMUL condition with purely random operands was approximately 1 in $5*10^{20}$, or 1 in 500 million trillion!

before a full-chip model was available. As described below, integrating and debugging all the logic and microcode needed to produce even a minimally functional full-chip model was a major undertaking; it took more than six months from the time we started until we had a "mostly functional" IA-32 machine that we could start to target for aggressive full-chip testing. Because we had the CTEs, we were able to start testing as soon as there was released code in a particular unit, long before we could have even tried to exercise it at the full-chip level.

Even after we had a full-chip model, the CTEs essentially **decoupled validation** of individual unit features from the health of the full-chip model. A blocking bug in, for example, the front end of the machine did not prevent us from continuing to validate in other areas. In actual fact, we rarely encountered this kind of blockage because our development methodology required that all changes be released at cluster level first, and only when they had been validated there did we propagate them to the full-chip level. Even then, we required that all full-chip model builds pass a mini-regression test suite before they could be released to the general population. This caught most major cross-unit failures that could not be detected at the CTE level.

One measure of the success of the CTEs is that they caught almost 60% of the bugs found by dynamic testing at the SRTL level. Another is that, unlike the Pentium Pro processor and some other new microarchitecture developments, the Pentium 4 processor never needed an SRTL "get-well plan" at the full-chip level where new development is halted until the health of the full-chip model can be stabilized.

POWER REDUCTION VALIDATION

From the earliest days of the Pentium 4 processor design, power consumption was a concern. Even with the lower operating voltages offered by P858, it was clear that at the operating frequencies we were targeting we would have difficulty staying within the "thermal envelope" that was needed to prevent a desktop system from requiring exotic and expensive cooling technology. This led us to include two main mechanisms for active power reduction in the design: *clock gating* and *thermal management*. Each of these is discussed in other papers in this issue of the *Intel Technology Journal*. Each presented validation challenges—in particular, clock gating.

Clock gating as a concept is not new: previous designs have attempted to power down discrete structures such as caches when there were no accesses pending. What was different about the Pentium 4 processor design was the extent to which clock gating was used. Every unit on the chip had a power reduction plan, and almost every Functional Unit Block (FUB) contained clock gating logic. In all, there were several hundred unique clock gating conditions identified, and each one of them needed to be validated from several different perspectives:

1. We needed to verify that each condition was implemented as per plan and that it functioned as originally intended. We needed to verify this not once, but continually throughout the development of the Pentium 4 processor, as otherwise it was possible for power savings to be eroded over time as an unintended side effect of other bug or speedpath fixes. We tackled this problem by constructing a master list of all the planned clock-gating features, and writing checkers in *proto* for each condition to tell us if the condition had occurred and to make sure that the power down had occurred when it should have. We ran these checkers on cluster regressions and low-power tests to develop baseline coverage, and then wrote additional tests as necessary to hit uncovered conditions.
2. While establishing this coverage, we had to make sure that the clock-gating conditions did not themselves introduce new logic bugs into the design. It is not hard to imagine all sorts of nightmare scenarios: unit A is late returning data to unit B because part of A was clock gated, or unit C samples a signal from unit D that is undriven because of clock gating, or other variations on this theme. In fact, we found many such bugs, mostly as a result of (unit-level) design validation or full-chip microarchitecture validation, using the standard set of checkers that we employed to catch such implementation-level errors. We had the ability to override clock gating either selectively or globally, and we developed a random power down application programming intereface (API) that could be used by any of the validation teams to piggyback clock gating on top of their regular testing. Once we had developed confidence that the mechanism was fundamentally sound, we built all our SRTL models to have clock gating enabled by default.
3. Once we had implemented all the planned clock-gating conditions, and verified that they were functioning correctly, we relied primarily on measures of clock activity to make sure that we didn't lose our hard-won power savings. We used a special set of tests that attempted to power down as much of each cluster as possible, and collected data to see what percentage of the time each clock in the machine was toggling. We did this at the cluster level and at the full-chip level. We investigated any appreciable increase in clock activity from model to model, and made sure that it was explainable and not due to designer error.
4. Last, but by no means least, we tried to make sure that the design was cycle-for-cycle equivalent with clock gating enabled and disabled. We had established this as a project requirement, to lessen the likelihood of undetected logic bugs or performance degradation caused by clock gating. To do this, we developed a methodology for *temporal divergence*

testing, which essentially ran the same set of tests twice, with clock gating enabled and disabled, and compared the results on a cycle-by-cycle basis.

We organized a dedicated Power Validation team to focus exclusively on this task, and they filed numerous bugs as a result of their pre-silicon validation (we filed "power bugs" whenever the design did not implement a power-saving feature correctly, whether or not it resulted in a functional failure). The results exceeded our fondest expectations: not only was clock gating fully functional on A-0 silicon, but we were able to measure approximately 20W of power saving in a system running typical workloads.

Full-chip Integration and Testing

With a design as complex as the Pentium 4 processor, integrating the pieces of SRTL code together to get a functioning full-chip model (let alone one capable of executing IA-32 code) is not a trivial task. We developed an elaborate staging plan that detailed just what features were to be available in each stage and phased in this plan over a 12-month period. The Architecture Validation (AV) team took the lead in developing tests that would exercise the new features as they became available in each phase, but did not depend upon any as-yet unimplemented IA-32 features. These tests were throwaway work—their main purpose was to drive the integration effort, not to find bugs. Along with these tests we developed a methodology which we called *feature pioneering*: when a new feature was released to full-chip for the first time, a validator took responsibility for running his or her feature exercise tests, debugging the failures, and working with designers to rapidly drive fixes into graft (experimental) models, thereby bypassing the normal code turn-in procedure, until an acceptable level of stability was achieved. Only then was the feature made available for more widespread use by other validators. We found that this methodology greatly speeded up the integration process. It also had a side effect: it helped the AV team develop their full-chip debugging skills much more rapidly than might otherwise have occurred.

Once a fully functional full-chip SRTL model was available, these feature pioneering tests were discarded and replaced by a new suite of IA-32 tests developed by the AV team, whose purpose was to fully explore the architecture space. Previous projects up to and including the Pentium Pro processor had relied on an "ancestral" test base inherited from past projects, but these tests had little or no documentation, unknown coverage, and doubtful quality (in fact, many of them turned out to be bug tests from previous implementations that had little architectural value). We did eventually run the "ancestral" suite as a late cross-check, after the new suite had been run and the resulting bugs fixed, but we found nothing of consequence as a result.

Coverage-Based Validation

Recognizing the truth of the saying: "If it isn't tested, it doesn't work" we attempted wherever possible to use coverage data to provide feedback on the effectiveness of our tests and tell us what we had and had not tested. This in turn helped direct future testing towards the uncovered areas. Since we relied very heavily on direct random test generators for most of our microarchitectural testing, coverage feedback was an absolute necessity if we were to avoid "spinning our wheels" and testing the same areas over and over again while leaving others completely untouched. In fact, we used the tuple of cycles run, coverage gained, and bugs found as our first-order gauge of the health of the SRTL model and its readiness for tapeout.

Our primary coverage tool was Proto from Intel Design Technology, which we used to create coverage monitors and measure coverage for a large number of microarchitecture conditions. By tapeout we were tracking almost 2.5 million unit-level conditions and more than 250,000 inter-unit conditions, and we succeeded in hitting almost 90% of the former and 75% of the latter. For the conditions that we were unable to hit prior to tapeout, we made sure that they were scattered throughout the entire coverage space and not concentrated in a few areas; and we also made sure that the System Validation (SV) team targeted these areas in their post-silicon validation plans. We also used Proto to instrument several thousand multiprocessor memory coherency conditions (combinations of microarchitecture states for caches, load and store buffers, etc.), and, as mentioned above, all the clock-gating conditions that had been identified in the unit power reduction plans. We used the Pathfinder tool from Intel's Central Validation Capabilities group to measure how well we were exercising all the possible microcode paths in the machine. Much to our surprise, running all of the AV test suite yielded coverage of less than 10%; further analysis revealed that many of the untouched paths involved memory-related faults (e.g., page fault) or assists (e.g., A/D bit assist). This made sense, as the test writers had (reasonably enough) set up their page tables and descriptors so as to avoid these time-consuming functions (at SRTL simulation speeds, every little bit helps!), but it did reinforce the value of collecting coverage feedback and not just assuming that our tests were hitting specified conditions.

POST-SILICON VALIDATION

As soon as the A-0 silicon was available, validation moved into the "post-silicon" phase. In post-silicon validation, the processor is tested in a system setting. Validation in this setting concentrates not only on the processor, but its interaction with the chipset, memory system, and peripherals. In this environment, the testing is done at real-time processor speeds, unlike the simulation environment that must be used prior to the arrival of the actual silicon. This is good news for test coverage, as much longer and more complex tests can be run in real-time, but it is bad news for debugging. In the SRTL simulator, all of the internal signals of the processor are available for inspection, but in the actual silicon, the primary visibility is from the transactions on the processor system bus. A significant effort went into preparing for the availability of the A-0 processor silicon. Hardware engineering teams developed and constructed validation platforms to provide execution and test vehicles for the processor silicon. The SV team assigned engineers to learn the microarchitecture of the processor and develop specific tests for the silicon. The Compatibility Validation (CV) team constructed an elaborate test infrastructure for testing the processor silicon using industry-standard operating systems, application software, and popular peripheral cards. The Circuit Marginality Validation (CMV) team prepared a test infrastructure for correlating tester schmoo plots with actual operational characteristics in systems capable of running standard operating systems as well as SV tests. All of these preparations were completed prior to the actual receipt of the A-0 processor silicon such that testing could proceed without delay as soon as the first silicon samples arrived.

Arrival of First Silicon

Systems developed for post-silicon validation included uniprocessor desktop systems, dual-processor workstation boards, 4MP server boards, and 4MP system validation platforms that include extensive test assistance circuitry (although the Pentium 4 processor is a uniprocessor product, we have found that certain types of *multiprocessor* testing can be good at exposing *uniprocessor* bugs). A few of each of these systems were available in the Pentium 4 processor system validation lab a few weeks prior to the arrival of first silicon samples. Within a few days after receiving the initial samples of A-0 processor silicon, we had successfully booted a number of operating systems including MS- DOS*, MS Windows* 98, MS Windows NT* 4.0, and Linux*.

The most complex and flexible validation platform was the 4MP system validation platform. This platform included the following key features:

- logic analyzer connectors
- SV hooks card that permits direct stimulus of FSB signals
- a software-controllable clock board that permits setting the processor system bus frequency in 1MHz steps
- software-controllable voltage regulators for both the CPU and chipset components
- built-in connectors for the In-Target Probe (ITP) debugging port
- four PCI hublink boards to support a large number of synthetic I/O agents
- sockets for the processor and chipset component silicon

*Other brands and names are the property of their respective owners.

System Validation

In parallel with the hardware system design, a team of System Validation (SV) engineers was assembled from a small core of experienced system validators. Learning from previous SV experiences, the team was assembled early to provide sufficient time for the engineers to learn the microarchitecture of the Pentium 4 processor and to develop an effective test suite. The team was also chartered with improving the effectiveness of system validation. A variety of test strategies, environments, and tools were developed to meet the challenge of accelerating the post-silicon validation schedule while achieving the same or a higher level of test coverage. The SV organization comprised a number of teams that targeted major CPU attributes:

- architecture—including the Instruction Set Architecture (ISA), floating-point unit, data space, and virtual memory
- microarchitecture—focusing on boundary conditions between microarchitectural units
- multi-processor—focusing on memory coherency, consistency, and synchronization

Different test methodologies were developed to test the various processor attributes. SV methodologies and test environments include

- Random Instruction Test (RIT) generators, which are highly effective for ISA testing, especially the interactions between series of instructions
- directed (focused) tests
- directed random tests (algorithmic tests with random code blocks inserted strategically)
- data space (or data domain) tests for testing boundary and random floating-point operands

SV tests are developed to run directly on the processor without an operating system run-time environment. Due to this, and the fact that the full test source is available and understood by the team, SV tests are relatively straightforward to debug in the system environment.

Random Instruction Testing

An especially effective method for testing the interactions between sequences of instructions is the Random

Instruction Test (RIT) environment. It is not mathematically feasible to even test just the possible combinations of a single instruction with all possible operands and processor states. Add to this the possibility of virtually limitless combinations of instruction sequences and it becomes clear that a systematic and exhaustive test strategy is wholly impractical. A practical and effective alternative is to construct an RIT environment. An RIT environment works in this way:

- The RIT tool generates a highly random sequence of instructions, sometimes described as a series of instructions that no sane programmer would ever devise.
- The instructions are presented in sequence to an architectural simulator, which constructs a memory image of the processor state (whenever memory is affected by a store, a push, or an exception frame).
- Once the test generation is concluded, the resulting test object and memory image are saved.
- The test is loaded onto the SV platform and executed.
- Following the conclusion of the test, the SV platform memory is compared against the memory image obtained from the architectural simulator. If the images match, the test passes; otherwise it fails. Another failure possibility is a “hang;” for example, the processor may experience a livelock condition and fail to complete the test.

A number of key requirements drove the development of such a test environment for the Pentium 4 processor:

- The first one was the ability to fully warm up the very long Pentium 4 processor pipeline. This is particularly difficult in an RIT environment, as truly random instruction combinations tend to cause frequent exceptions or other control-flow discontinuities. Typical RIT tools available before the new tool was developed would typically encounter a pipeline hazard within 3 to 20 instructions on average. This could result in missing bugs that might exist in actual application or operating system software.
- The second one was the ability to avoid “false” failures, e.g., failures occurring due to undefined processor states or other differences between an architectural simulator and the actual processor silicon. This is an extremely important feature, as a high false failure rate will limit the useful throughput of such a tool. Every failure must be examined, whether false or real (otherwise, how does one know if a real failure has occurred?).
- The third one was the ability of the RIT environment to fully propagate the processor state to the memory image file without unduly affecting the randomness of the instruction stream. Lacking this feature it is possible to miss failures due to the tendency of RIT tools to frequently overwrite state, thus potentially hiding the failing signature of a bug.
- The fourth one was the ability of the tool to greatly increase RIT throughput. The new tool increased the throughput by a factor of 100 over existing tools. This was essential to find rarely occurring or highly intermittent bugs.

The new tool, known as Pipe-X (for Pipeline Exerciser) proved to be extremely effective, logging the most bugs of any SV test environment or test suite. It has effectively a zero false failure rate, without which the high throughput would prove to be unmanageable from a debugging standpoint. For a given processor stepping that requires production qualification, one billion tests (each 10,000 instructions in length) are executed in approximately eight weeks. To date, approximately 10 billion RIT tests have been executed on the Pentium 4 processor, compared with the approximately 10 million RIT tests that have been executed on all versions of the Pentium® II and Pentium® III processor families. Pipe-X has been found to be effective in finding both architectural and microarchitectural logic bugs.

Focused SV Testing

We used directed or focused testing to complement RIT. It is important to perform algorithmic testing of major processor components. A comprehensive set of focused tests was available from the Pentium Pro, Pentium II, and Pentium III processor families. This test suite is known as the Focus Test Suite (FTS) and is particularly effective at finding cache bugs, Programmable Interrupt Controller (PIC) bugs, and general functional bugs. The focus test suite has been in continuous development for many years, and was effectively doubled in size to prepare for Pentium 4 processor validation. It has found almost the same number of bugs as Pipe-X.

Compatibility Validation

Although SV finds most post-silicon bugs (approximately 71%), and those bugs are the most straightforward to debug, it is vital to ensure that the new processor, chipset, and memory system works correctly with standard operating systems using a wide variety of software applications and peripheral cards. For this reason, an extremely elaborate Compatibility Validation (CV) laboratory was assembled. CV tests are designed to particularly stress interactions between the processor and chipset, concentrating on causing high levels of FSB traffic. The CV staff often work closely with Original Equipment Manufacturers (OEMs) to resolve problems sighted at the OEM and assist in performance validation by running standard benchmark workloads. CV tests also help to weed out software problems in BIOS. The CV team will see most of the bugs that the SV team uncovers, but due to the difficult nature of debugging in the CV environment, the bugs found in SV are resolved much more quickly.

Debugging in the System Environment

Debugging in the laboratory using actual Pentium 4 processor silicon installed in a validation platform or PC-like test vehicle is difficult in the extreme. The very best place to debug a processor bug is in the processor simulator, where all signals are available for scrutiny. Unfortunately, in the system environment almost no internal signals are visible to the system debugger. A suite of tools was developed for use in the Pentium 4 processor, using both architectural and microarchitectural features of the processor:

The In-Target Probe (ITP) consists of a scan-chain interface with the processor that connects to a host PC system. Using the ITP, the debugger can set breakpoints, examine memory and CPU registers, and start and stop processor program execution. This tool is helpful for interactive patching of test programs, for single-stepping program execution, and for loading small test fragments. In other words, for blatant functional bugs in the processor this tool is effective. However, many bugs only happen when the processor is running at full speed with multiple processors and threads executing, and frequently a bug will immediately disappear when inserting a breakpoint near the failure point.

Scan chain-based array dumps can be used to construct limited watch windows of a small set of select internal signals. This can be especially useful for identifying the signature of some bugs.

Logic analyzer trace captures of the processor system bus can be translated into code streams that can be run on a hardware-accelerated processor RTL model. Such a methodology is based upon forcing all instructions to be fetched on the bus due to periodic invalidation of processor caches.

Validation platform features permit the schmooing of voltage, temperature, and frequency to help accelerate the occurrence of circuit bugs. However, the most effective environment for debugging circuit problems is the semiconductor tester lab.

Due to the extremely complex and lengthy Pentium 4 processor pipeline, many bugs are extremely difficult to reproduce. Being able to capture such failures on a logic analyzer and subsequently running the resulting program fragment on a hardware-accelerated RTL model has time and again proven to be almost the only method for isolating highly intermittent bugs.

Debugging has historically been the primary limiter to post-silicon validation throughput, and despite significant improvements in debugging based on the use of the logic analyzer, it is usually on the critical path to production qualification.

BUG DISCUSSION

Comparing the development of the Pentium® 4 processor with the Pentium® Pro processor, there was a 350% increase in the number of bugs filed against SRTL prior to tapeout. The breakdown of bugs by cluster was also different: on the Pentium Pro processor [2] microcode was the largest single source of bugs, accounting for over 30% of the total, whereas on the Pentium 4 processor, microcode accounted for less than 14% of the bugs. For both designs, the Memory Cluster was the largest source of hardware bugs, accounting for around 25% of the total in both cases. This is consistent with data from other projects, and it indicates that for future projects we should continue to focus on preventing bugs in this area. We also determined that almost 20% of all the bugs filed prior to tapeout were found by code inspection.

We did a statistical study [3] to try to determine how the bugs came to be in the Pentium 4 processor design, so that we could improve our processes for preventing bugs from getting into future designs. The major categories were as follows:

- RTL Coding (18.1%)—These were things like typos, cut and paste errors, incorrect assertions (instrumentation) in the SRTL code, or the designer misunderstood what he/she was supposed to implement.
- Microarchitecture (25.1%)—This covered several categories: problems in the microarchitecture definition, architects not communicating their expectations clearly to designers, and incorrect documentation of algorithms, protocols, etc.
- Logic/Microcode Changes (18.4%)—These were bugs that occurred because: the design was changed, usually to fix bugs or timing problems, or state was not properly cleared or initialized at reset, or these were bugs related to clock gating.
- Architecture (2.8%)—Certain features were not defined until late in the project. This led to shoehorning them into working functionality.

Post-Silicon Bugs

An examination of where bugs have been found in the post-silicon environment reveals the following data:

- System Validation (71%)—The dominance of SV is intentional, as it is definitely the best environment in which to debug post-silicon bugs. A wide spectrum of logic and circuit bugs is found in this environment.
- Compatibility Validation (7%)—Although this team doesn't find as many bugs as SV, the bugs found are in systems running real-world operating systems and applications. Most of the bugs found in SV will also be seen in the CV environment.

- Debug Tools Team (6%)—The preponderance of bugs found by the debug tools team were found in the first few weeks after A-0 processor silicon arrived. This stems from the fact that getting the debug tools working is one of the earliest priorities of silicon validation.
- Chipset Validation (5%)—The chipset validation teams concentrate on bus and I/O operations, and the bugs found by this team reflect that focus: they are typically related to bus protocol problems.
- Processor Architecture Team (4%)—The processor architecture team spends much time in the laboratory once silicon arrives, examining processor performance and general system testing. This team also plays a central role in debugging problems discovered by the SV, CV, and other validation teams.
- Platform Design Teams and Others (7%)—This group includes the hardware design teams that develop and deploy validation and reference platforms, the processor design team, the pre-silicon validation team, and software enabling teams.

CONCLUSION

The Pentium® 4 processor was highly functional on A-0 silicon and received production qualification in only ten months from tapeout. The work described here is a major reason why we were able to maintain such a tight schedule and enable Intel to realize early revenue from the Pentium 4 processor in today's highly competitive marketplace.

ACKNOWLEDGMENTS

The results described in this paper are the work of many people over a multi-year period. It is impossible to list here the names of everyone who contributed to this effort, but they deserve all the credit for the success of the Pentium® 4 processor validation program.

REFERENCES

[1] Clark, D, "Bugs Are Good: A Problem-Oriented Approach To The Management Of Design Engineering," *Research-Technology Management*, 33(3), May 1990, pp. 23-27.

[2] Bentley, R., and Milburn, B., "Analysis of Pentium® Pro Processor Bugs," *Intel Design and Test Technology Conference*, June 1996. Intel Internal Document.

[3] Zucker, R., "Bug Root Cause Analysis for Willamette," *Intel Design and Test Technology Conference*, August 2000. Intel Internal Document.

AUTHORS' BIOGRAPHIES

Bob Bentley is the Pre-Silicon Validation Manager for the DPG CPU Design organization in Oregon. In his 17-year career at Intel he has worked on system performance evaluation, processor architecture, microprocessor logic validation and system validation in both pre- and post-silicon environments. He has a B.S. degree in Computer Science from the University of St. Andrews. His e-mail address is bob.bentley@intel.com

Rand Gray is a System Validation Manager for the DPG Validation organization in Oregon. He has more than 20 years of experience with microprocessors, including processor design, debug tool development, and systems validation. He has a B.S. degree in Computer Science from the University of the State of New York. His e-mail address is rand.gray@intel.com.

Managing the Impact of Increasing Microprocessor Power Consumption

Stephen H. Gunther, Desktop Platforms Group, Intel Corp.
Frank Binns, Desktop Platforms Group, Intel Corp.
Douglas M. Carmean, Desktop Platforms Group, Intel Corp.
Jonathan C. Hall, Desktop Platforms Group, Intel Corp.

Index words: Pentium® 4 processor, power, power simulation, thermal management, thermal design, heat-sinks

ABSTRACT

The power dissipation of modern processors has been rapidly increasing along with increasing transistor count and clock frequencies. At the same time, there is a growing disparity between the maximum power consumption of a processor and the "typical" power consumed by that processor; i.e., power consumed while running typical applications. This trend is the result of the significant increase in transistor count required to reach the desired peak performance targets.

Designing a processor with the intent of minimizing system costs, especially those arising from high power consumption, while retaining a high level of reliability requires attention at all stages of the design. In the case of the Pentium® 4 processor, the design team focused from the beginning on reducing power consumption without negatively impacting either the performance or reliability of the processor in any significant way. Many techniques, both innovative and pre-existing, were applied across the entire processor in an effort to eliminate unnecessary power consumption. The mass adoption of these techniques resulted in a significant reduction in both maximum and typical processor power dissipation.

INTRODUCTION

The total power dissipation of recently introduced, new-generation, microprocessors has been rapidly increasing, pushing desktop system cooling technology close to its limits. The Pentium 4 processor is the first new-generation IA-32 microarchitecture processor to significantly improve upon the historical IA-32 processor power trends. The power savings achieved in the design of the Pentium 4 processor will translate into lower cost systems, higher frequency processors, and improved manufacturing yield while maintaining the high level of reliability and quality for which Intel is known.

This paper outlines the guiding principles that were set in place when the Pentium 4 processor was first defined. We first describe an engineering process and tool chain that made the low-power aspects of the design visible and supported a feedback path to the design team. We also briefly touch on the key lessons that were learned in the design, validation, and debugging of clock gating and other power-conserving elements of the Pentium 4 processor.

We then present a processor power-monitoring and control mechanism that is entirely contained on the processor die. No off-chip hardware or software interaction is required to guarantee that a pre-determined temperature threshold is not exceeded during processor operation. The architecture of this thermal monitor control logic closely maps to the existing Advanced Configuration and Power Interface (ACPI) specifications. The monitoring and control mechanism consists of three separate but related functions: a temperature detection mechanism, a power reduction mechanism, and control and visibility logic. Each of these functions, and the implementation constraints, are described in detail in this paper.

PROCESSOR POWER TRENDS

The power dissipation of modern processors is rapidly increasing as both clock frequency and the number of transistors required for a given implementation grow. Figure 1 shows the power consumption trend of processors introduced by Intel over the past 15 years. As can be seen, the general trend is for maximum processor power consumption to increase by a factor of a little more than 2X every four years.

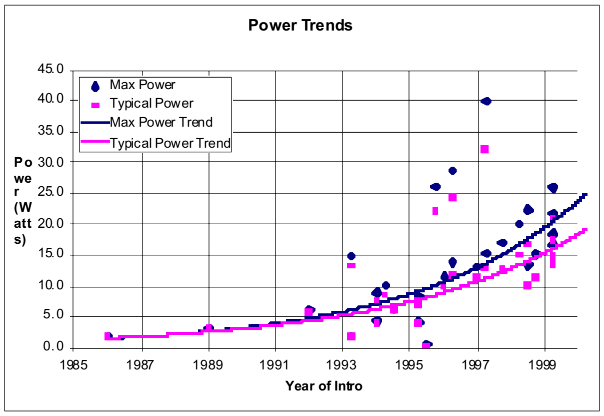


Figure 1: Trends in CPU power consumption

The second trend to note, also shown in Figure 1, is the increasing disparity between the maximum power consumption of a processor and the power consumed by that same processor while running more typical applications. For a typical Intel® processor introduced from 1996 onwards, the power consumed when running a synthetic high-power workload was 20% higher than the power consumed by the same processor while running a high-power section of a real application.

The disparity between maximum power and typical power consumed presents a particularly difficult problem to the system designer. The system must be designed to ensure that the processor does not exceed the maximum specified operating temperature, even when it is dissipating the maximum power. While designing an elaborate heat sink, or forcing more air through the system can usually meet this constraint, there is usually significant cost associated with more elaborate solutions and environmental regulations that limit system (fan) noises.

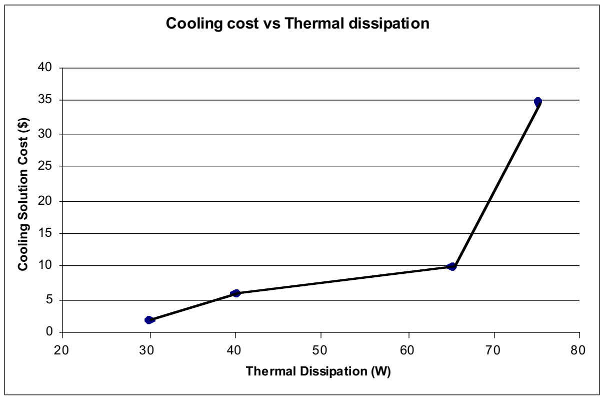


Figure 2: The cost of removing heat from a microprocessor

Figure 2 shows the relative implementation cost of various cooling solutions ranging from a simple extruded aluminum heat sink to a more elaborate heat pipe technology. It is important to observe that as power increases, there is a non-linear relationship between the cooling capabilities and the cost of the solution. This emphasizes the importance of limiting maximum power consumption to a specific envelope, one defined by the cost structure of the platform for which the processor is intended.

PROCESSOR FEATURE DEFINITION AND TRACKING

The Pentium 4 processor team started on the journey to a lower power design by defining power reduction features during the definition phase of the processor. The overall processor power reduction was to be achieved with a combination of architectural (i.e., thermal monitor) techniques and microarchitectural/circuit (i.e., clock-gating and low-power circuit) techniques. Once the power reduction features were defined, it became important to track the actual power savings of each microarchitectural feature in a manner synergistic with microprocessor development techniques.

A well-defined infrastructure was put in place to track various aspects of the Pentium 4 processor power reduction effort. The infrastructure included a series of interactive reviews, indicators, and regression tests. The first review was put in place as a checkpoint prior to the start of code development. This review examined the basic power plan and identified the specific set of power-saving features. The output of the review was a common format, easy to read summary of the power-reduction plan.

An internal design indicator, dubbed the Wattmeter, see Figure 3, was developed to track the implementation status and the power savings achieved by each feature. The Wattmeter was used to track progress toward specific power goals. The data for the Wattmeter were based on the unit-level power reduction plans, RTL coding information, and the specific circuit style utilized for each Functional Unit Block (FUB). The relative impact of each feature was combined with the expected power consumption of the FUB in question, and an architectural-level activity factor was applied to yield an estimate of the power saved by that feature in Watts. As specific features were coded, the impact of those features on power consumption was added to the Wattmeter to influence the power-savings indicator.

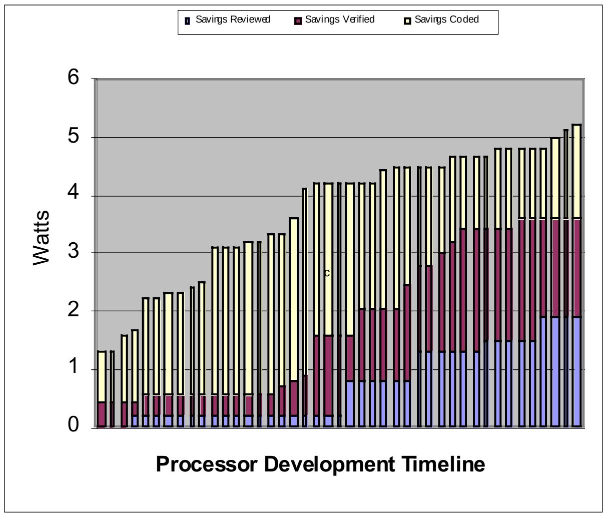


Figure 3: The Wattmeter

The Pentium 4 processor team identified more than 400 individual power-reduction opportunities that were ultimately implemented in the processor. According to projections from the various tools, some of these features resulted in a significant reduction in power, while most had a smaller impact. Power reduction features implemented in roughly 20% of the FUBs accounted for 75% of the total power savings achieved on the processor.

ARCHITECTURAL-LEVEL POWER SIMULATION

A new power estimation tool was developed to facilitate the evaluation of various power reduction features prior to the availability of a fully featured RTL model. This tool, referred to in this paper as the Architectural-Level Power Simulator (ALPS), allowed the Pentium 4 processor team to profile power consumption at any hierarchical level from an individual FUB to the full chip. The ALPS allowed power profiling of everything, from a simple microbenchmark written in assembler code, to application-level execution traces gathered on real systems.

At the most abstract level, the ALPS methodology consists of combining an energy cost associated with performing a given function with an estimate of the number of times that the specific function is executed. The energy cost is dependent on the design of the product, while the frequency of occurrence for each event is dependent on both the product design and the workload of interest. Once these two pieces of data are available, generating a power estimate is simple: multiply the energy cost for an operation (function) by the number of occurrences of that function, sum over all functions that a design performs, and then divide by the total amount of time required to execute the workload of interest.

The difficulty comes in gathering each of the required pieces of data. The benefit of being able to estimate power consumption is highest early in the design, yet detailed data on event frequency and energy cost are often not available. Therefore, it is often necessary to make significant approximations based on the data that are available.

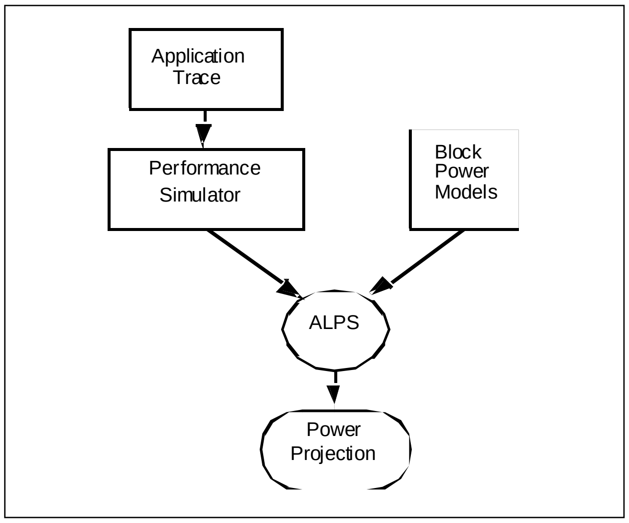


Figure 4: The ALPS flow

The ALPS methodology, flow charted in Figure 4, utilizes data from several sources. First of all, a high-level description of the design is used to identify which functional events are likely to have a significant impact on chip power consumption. This behavioral description is coupled with historical data on the power consumption of different types of functional blocks, as well as designer estimates, to approximate the energy cost of performing each of the functions associated with a given logic block. This results in a power model for each logic block of interest. We can then measure the frequency of occurrence for each of these events by utilizing the statistics generated by an architectural-level performance simulator for the design. The actual power estimate for a given instruction sequence is then made by ALPS as it executes the equation shown in Figure 5.

$$\text{Total Power} = \frac{\sum_{\text{Event}_1}^{\text{Event}_n} (\text{Energy}_n \times \text{Occurrences}_n)}{\text{Execution Time}}$$

Figure 5: Equation for total power

DEVELOPING THE BLOCK POWER MODELS

In order to focus model development efforts, it is necessary to understand the behavior of the processor to a level of detail sufficient to identify which functional events have the greatest impact on total power consumption. By initially focusing only on those events that were expected to result in significant power consumption, we were able to quickly generate a simple power model, which allowed for incremental improvements as additional data became available. To clarify, it is known that each access to a large memory array (such as an on-chip cache) will cause the simultaneous transition of many address and data lines and have a noticeable impact on total power consumed. Conversely, the power associated with a small state machine that controls those cache accesses may have much less of an impact.

For similar reasons, events that are expected to occur very infrequently are also less important to model. As a further example, based on the frequency of occurrence, developing a power model for a logic block that generates a physical memory address for each load instruction is probably far more important than having a power model for a logic block that detects floating-point exceptions.

From a data collection standpoint, the functional-block-level information can be broken into two parts: information used to determine the activity level of a particular logic block and data to facilitate estimating the energy cost of each type of activity.

Each logic block in the design may perform one or more distinct logical functions. The activity level of the block as a whole is dependent on how often these functions are performed, the percentage of the logic that is associated with each function, and the length of time it takes to perform the function.

In the case of the Pentium 4 processor, we initially considered each unit to have five to eight key functions, and then added to this list, as additional design information became available. The initial list of activities was first based on a general understanding of the microarchitecture; it was later refined as the high-level processor definition code as the RTL for the block took form. Typically, extensive interaction with unit architects and designers was needed to clarify the events and activities key to each logic block.

POWER VALIDATION

Clock gating refers to activating the clocks in a logic block only when there is work to be done, and it is one of the key power-saving techniques employed on the Pentium 4 processor. When performing clock gating on a massive scale, two validation concerns arise. With overaggressive clock gating, logic failures can arise, where a block should have been awake, but either did not turn on quickly enough, or is turned off too soon. On the other hand, conservative clock gating will not disrupt correct functionality, but will result in wasted power. Functional checkers such as an architectural simulator do not report on such failures.

As one might anticipate, not all power-saving features have equivalent value. Therefore, a two-pronged approach was taken to finding these logic problems. One was fine-grained, carefully focusing on just the key power-saving features. The other was coarse-grained, examining all gated clocks for abnormal activity. For each approach, specific tools and methodologies were created to automate the process of finding these power-wasting logic problems. Use of these tools continued throughout the entire design process.

To uncover logical failures associated with clock gating, each Pentium 4 processor unit's Design Validation (DV) Test Plan was reviewed to ensure that the validator addressed the power down corner cases, and that such cases are included in the overall validation coverage figures.

THERMAL MONITOR OVERVIEW

Implementing a traditional thermal solution that accommodates the maximum power consumption of a leading-edge microprocessor like the Pentium 4 processor would have a significant impact on the system cost. To reduce this cost but retain a high level of reliability, an enhanced version of an existing mechanism used in the mobile computing segment, processor power modulation, was used in the Pentium 4 processor.

The processor power modulation mechanisms employed in mobile systems have taken two forms, both of which require the cooperation of external logic. The first mechanism involves slowly reducing the processor clock frequency, typically from its maximum supported frequency down to a lower frequency. The second mechanism involves the modulation of the processor STOPCLOCK signal (the pin named STPCLK#, while asserted, has the effect of stopping the clock to many internal elements of the processor). Since power consumption is proportional to operating frequency, both mechanisms have a similar effect on the power consumed by the processor.

The external logic that controls the power modulation of the processor could be activated by numerous stimuli, including detection of high processor or system temperatures, detection of low available battery power, or simply by a user selecting a low-power operating mode (with the goal of extending battery life).

In the case of a desktop computing system, one of the key constraints is the requirement to control the operating temperature of the processor. This requires being able to accurately measure the temperature of the processor silicon. Unfortunately, this is difficult to achieve with external temperature sensors. There is a significant delay between the time at which the processor silicon reaches a given temperature and the time at which an external temperature sensor notices the temperature change. Several solutions have previously been pursued, varying from attaching the temperature sensors to heat sinks, to the processor package, or mounting it underneath the processor. Each solution has the same problem: reliable high-volume manufacturing is difficult.

More recently, portions of the thermal sensor (e.g., the thermal diode) have been located on the processor die. Even this approach has clear limitations. The temperature from one corner of the die to another can vary by a significant amount, so understanding sensor placement on the die is critical. Additionally, the rate at which the die temperature can change is increasing to the point that the currently available thermal sensor interface logic is too slow to allow reliable die temperature measurement or closed loop thermal control.

With the Pentium 4 processor, the objective was to enable accurate control of processor die temperature. The solution chosen integrates all portions of the power modulation mechanism including temperature detection through power control. When this feature is enabled, the processor is capable of operating with no further software intervention. In anticipation that software will eventually be required to control processor power dissipation, the architecture of the thermal monitor control logic has been created in such a way that it closely maps to the existing Advanced Configuration and Power Interface (ACPI) specification and software.

The thermal monitor architecture implemented on the Pentium 4 processor can be described as three separate but related functions: a mechanism for determining temperature, a mechanism for reducing the power consumption of the processor, and a means of controlling and providing visibility into each of these mechanisms. Each of these three functions is described in detail in the following sections.

TEMPERATURE DETECTION MECHANISM

All integrated circuits are designed to operate reliably within a defined temperature range. Outside of this range, there is no assurance that the integrated circuits will continue to function correctly. The die temperature at any given point in time is a function of the power consumed by the device (both at a given instant in time and in the relatively recent past), the collective thermal coefficient from the die through the heat sink, and the ambient environmental conditions.

The temperature at any given point on the die can be measured with the use of a diode and a precise current source. The voltage drop across a diode is dependent on both the current flowing through the diode and the temperature of the diode. By supplying a constant current, and measuring the voltage drop across a diode, we can get a reasonably accurate measurement of the temperature at which the diode is currently operating. By comparing this voltage to a reference point, we can determine when the temperature of the diode (and hence the portion of the die that contains the diode) is just below the maximum specified operating temperature. This is the only temperature with which we are concerned.

There are a couple of key factors that significantly impact the accuracy of such a thermal sensor. The characteristics of both the diode used as the thermal sensor and the transistors used to create the current source are dependent on the specific parameters of the manufacturing process. Many of the process parameters change slightly from one wafer to another or from one area on the wafer to another, affecting the temperature recorded by the thermal sensor.

The second factor impacting the accuracy of the thermal sensor is the fluctuation in the processor operating voltage (measured on the die rather than at the pins of the processor package). This noise can result in the thermal sensor comparator (which determines whether the die temperature has reached the maximum operating temperature) signaling that the die is too hot, when in fact it has not yet reached the critical temperature. Alternatively, this noise could also cause the comparator to incorrectly signal that the die is below the critical temperature.

The Pentium 4 processor implements mechanisms to account for both of these sources of error in the output of the thermal sensor. In the case of a microprocessor, the power consumed is a function of the application being executed. In a large design, different functional blocks will consume vastly different amounts of power, with the power consumption of each block also dependent on the workload. While the heat generated on a specific part of the die is dissipated to the surrounding silicon, as well as the package, the inefficiency of heat transfer in silicon and between the die and the package results in temperature gradients across the surface of the die. Therefore, while one area of the die may have a temperature well below the design point, another area of the die may exceed the maximum temperature at which the design will function reliably. Figure 6 is an example of a simulated temperature plot of the Pentium 4 processor.

As a result of the cross die temperature variations; it is very important that the temperature detection mechanism (the integrated thermal sensor in the case of the Pentium 4 processor) be located at the hottest spot on the die. As can be seen from Figure 6, there are clearly optimal locations for placement of the thermal sensor.

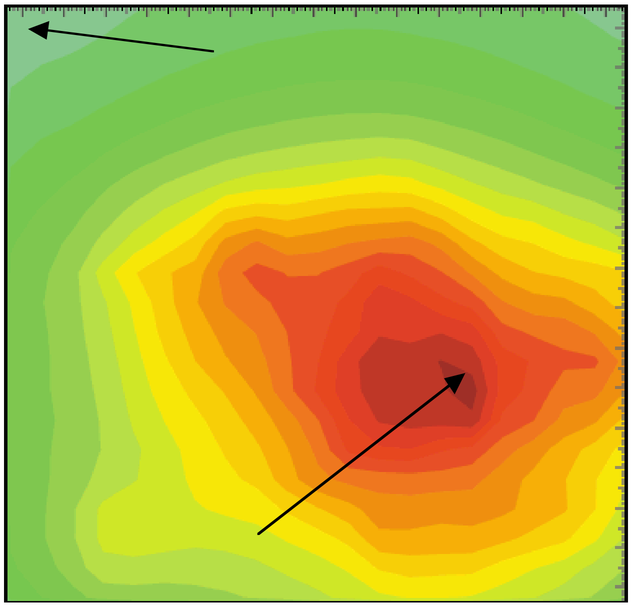


Figure 6: Simulated thermal plot of processor die

Because the hottest location on the die may change from application to application, it is important to confirm theoretical thermal maps of the die with actual measured data taken while various types of applications are executing. By evaluating the thermal maps for several classes of applications, it is possible to confirm the optimal location for the thermal sensor. Evaluation of these thermal maps also guides the selection of appropriate guard bands to be applied to the thermal sensor trip temperature. These guard bands are intended to ensure that as long as the temperature measured by the thermal sensor is below the maximum reliable operating temperature, there will be no location on the die that exceeds the maximum reliable operating temperature.

POWER REDUCTION MECHANISM

Once it has been determined that the die temperature is approaching the critical point, a mechanism is needed to quickly reduce power consumption, causing a drop in temperature. There are several key constraints in the design of this mechanism.

First, the latency between critical temperature detection and power reduction should be low. In this case, low latency refers to periods on the order of 100's of microseconds. Reaction times significantly longer than this would allow the die temperature to potentially reach a point at which it no longer operates reliably.

Second, the mechanism should be efficient. Here, efficiency refers to the ratio between power reduction and performance loss. An ideal mechanism results in a power vs. performance curve that is linear and crosses both axes at 0. In other words, if the power modulation mechanism results in a 10% performance loss while operating, it would also provide a 10% reduction in power consumption. Note an ideal relationship is only possible if frequency is the only variable.

Finally, the mechanism should add little or no cost to the design. Costs include those associated with die size, validation, platform impact, and risk.

After evaluation of a number of potential options, the Pentium 4 processor design team chose a mechanism that utilizes the existing architectural low-power logic (the StopClock architecture). The chosen mechanism essentially provides an internal STOPCLOCK request to the processor core.

This STOPCLOCK request results in the clock signal to the bulk of the processor logic being stopped for a short time period. While this clock signal is stopped, the power consumption of the processor is reduced to a small fraction of the maximum processor power consumption. Because the STOPCLOCK request is a relatively high priority interrupt, the delay between the request and the resulting power decrease is relatively short, typically much less than 1 microsecond.

In order to minimize any potential impact to the platform, the time period during which the clock is stopped is kept small. The design target limits the total time during which the processor is not executing useful code to a few microseconds. This includes both the time the clock is actually stopped and the overhead associated with stopping and restarting the clock signal.

THERMAL MONITOR CONTROL

The behavior of the power modulation mechanism can be controlled with an enable bit in a model-specific register. When enabled, the power modulation mechanism is automatically invoked whenever the thermal sensor indicates that the die is hot. The power modulation mechanism remains engaged until the die temperature drops below the critical value. While the default condition has this bit set to ”disabled”, it is required that the normal usage model would enable the thermal monitor functionality during the initialization process, and leave it enabled for as long as the system is powered on. This usage model provides the most robust processor thermal solution, as the processor can protect itself from most causes of overheating without any interaction by system hardware or software. The thermal monitor mechanism can also be invoked via the ACPI compatibility registers (see the section on ACPI interaction for details).

THERMAL MONITOR VISIBILITY

Although the thermal monitor mechanism implemented on the Pentium 4 processor can be configured to engage automatically and transparently, it may still be desirable to signal the thermal monitor state to the system hardware and operating system. In the Pentium 4 processor implementation, this signaling is provided via three means.

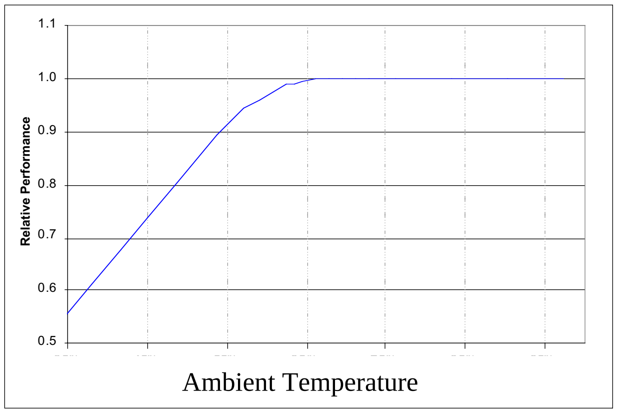


Figure 7: Example simulated system design curve

On the hardware side, an output signal reflects the state of the thermal sensor comparator. This signal is asserted while the thermal sensor indicates that the die temperature is at the maximum operating temperature, and is de-asserted while the temperature is below this point. This signal could be used by system hardware to take some action when the die temperature reaches the critical temperature. Note, however, if the closed loop thermal control is disabled, any cooling oriented action must be effective very quickly to prevent the processor from overheating and failing (see the section on the power-reduction mechanism). For example, using this signal to turn on an additional fan in the system would not be a sensible solution as the processor temperature could easily exceed its maximum operating temperature before the cooling effect of the fan is noticed.

On the software side, there are two, model-specific register-based, status bits of interest. The first reflects the state of the thermal sensor comparator. This information is identical to that provided by the output signal. The second bit is a "sticky" bit, which is set the first time the thermal sensor reaches the critical temperature, and it must be explicitly cleared by software or through a processor reset. An example use of the "sticky" bit would be for diagnostic software to determine if the processor has ever reached the critical temperature, which could be used as an indicator that the thermal solution performance has changed.

The final visibility mechanism consists of the ability to generate an interrupt request whenever there is a change in the output of the thermal sensor comparator. These interrupts can be generated in either direction; i.e., an interrupt can be generated when the thermal sensor output transitions from the "not hot" state to the "hot" state, and/or when the thermal sensor output transitions from the "hot" state to the "not hot" state. Each of these interrupts can be individually enabled or disabled.

PERFORMANCE

One of the primary themes behind the development of the described thermal monitor mechanism is the ability to reduce system thermal design costs without a perceivable impact on performance. Because the thermal monitor mechanism could impact performance if the processor reaches the critical temperature, it is valuable to understand how often, and for how long, the critical temperature could be reached while running real application code. These data allow system designers to design a solution that optimally balances system cost and the thermal performance required.

The performance impact resulting from the thermal monitor is dependent on both the processor power consumption and the thermal solution. By generating a curve of thermal monitor performance impact vs. system thermal capability, the system designers can determine the design point that is optimal for their target market.

During the development process of the Pentium 4 processor, the Architectural-Level Power Simulator (ALPS) was used to project the power consumption of a range of applications. By using the power-consumption projections of the ALPS, along with the expected thermal characteristics of the Pentium 4 processor package, it was possible to project the temperature of the Pentium 4 processor die at a given point in time while running the applications of interest.

The resulting temperature vs. time data could then be used to project when the processor would reach the critical temperature. Using the characteristics of the thermal monitor mechanism, along with the package characteristics, it is possible to project how long the thermal monitor mechanism would remain active. The process described was automated and was used to generate curves of processor performance vs. system thermal design capability. Figure 7 shows an example of one of these curves. As can be seen from Figure 7, the thermal monitor mechanism implemented has the potential for significantly reducing the system thermal design point, without perceivably impacting processor performance.

INTERACTION WITH THE ADVANCED CONFIGURATION AND POWER INTERFACE SPECIFICATION

The Advanced Configuration and Power Interface (ACPI) specification defines a hardware and software environment that allows operating system software complete visibility and control of system configuration and power management. The specification describes a set of valid processor operating states and the allowable transitions between them. The upper four states defined for the processor are as follows:

1. C0—normal operation
2. C1—a low-power, low-latency state that assumes no support from chipset logic that retains all cached context

3. C2—a lower-power, slightly longer latency state than C1 that requires chipset support but still retains cached context
4. C3—a still lower power, longer latency state that also requires chipset support but one in which the cached context may be lost

Systems based on the IA-32 architecture will typically map the use of the HALT (HLT) instruction to the C1 state, the STOPCLOCK assertion to C2, and Deep Sleep (removal of the processor clock input signal) operation to the C3 state.

A documented sub-mode of the ACPI, C0 state is known as Clock Throttling (the thermal control functionality on the Pentium 4 processor would map to this sub-mode of the ACPI spec). In this mode, the operating system accesses logic to assert the STOPCLOCK signal with some predetermined duty cycle prior to the Pentium 4 processor, this logic had been resident in the chipset). The term "duty cycle" is used to refer to the characteristics of the signal applied by the chipset to the processor's STOPCLOCK pin in order to reduce processor power dissipation.

The register that is defined to enable and configure Clock Throttling is named the Processor Control register (P_CNT) by the ACPI specification. This 32-bit register has bits defined to both set the Clock Throttle (power control) duty cycle and force the thermal control to begin. The actual width and offset within P_CNT of the duty cycle field can be configured by a system developer. The Pentium 4 processor has implemented this P_CNT register in internal Model Specific Register (MSR) space. The three duty cycle bits implemented in the Pentium 4 processor's control register give software the ability to define seven levels of power control, with one value (0) reserved.

The incorporation of the P_CNT register in the processor provides the operating system with the ability to perform thermal control on a per processor basis even when there are multiple processors in a system. The Pentium 4 processor does not support Multi-Processor (MP) system configurations; however, there will be future MP capable IA-32 processors based on this same microarchitecture targeted at the server and workstation market. It should be noted that, to date, chipsets have only implemented a single set of ACPI Clock-Throttling registers, and that chipsets have a single STOPCLOCK pin. The net impact is that a per processor Clock Throttle solution does not currently exist for MP systems. Incorporation of a P_CNT register into the processor solves this issue without requiring the addition of multiple STOPCLOCK pins in the chipset.

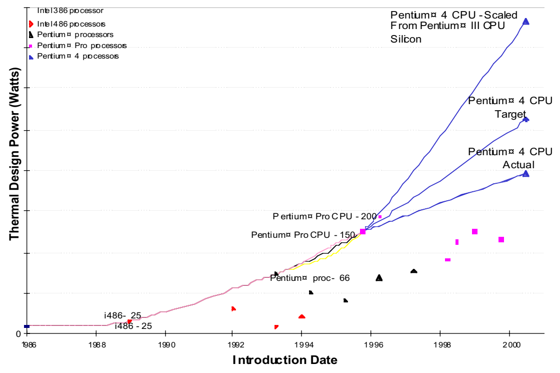


Figure 8: Pentium® 4 processor power with historical trends

It is interesting to observe that the Thermal Monitor and Clock Throttle functions, although similar and intended to cooperate/inter-operate, are designed to allow control of fundamentally different system parameters. The Thermal Monitor is intended to very closely control the processor die temperature, ensuring that the processor temperature remains within the specified range. The Clock Throttling defined by ACPI is intended to allow the operating system to modulate the processors' power dissipation in order to control the ambient temperatures that may impact other components within the system.

CONCLUSION

At the start of the Pentium 4 processor project, the design team formally committed themselves to lowering the processor power consumption by 20% from initial power projections . The team also committed to lower the thermal design point of the system by 40% without perceivably impacting application performance, while maintaining processor reliability. These commitments were met in the initial version of the Pentium 4 processor. For reference, Figure 8 depicts the level of power savings that the Pentium 4 processor achieved, superimposed on historical thermal design power data.

The power reduction achieved resulted largely from the extensive application of clock gating and unit power-down techniques. The addition of the thermal monitor feature enables what is essentially a processor that is capable of managing its own thermal profile to operate efficiently within almost all thermal environments.

ACKNOWLEDGMENTS

The authors thank the Pentium 4 processor design team for their significant effort to minimize the power consumption of the processor, with special thanks to Bindu Lalitha, Kimberly Weier, Bret Toll, Kan Chu, and Linda Rankin.

AUTHORS' BIOGRAPHIES

Stephen H. Gunther joined Intel in 1992. He has spent much of the last eight years in the Desktop Processor Group working on techniques for reducing the power consumption of Intel microprocessors. Steve received his B.S. degree in Computer Engineering from Oregon State University in 1987. He is currently involved in the development of future microprocessors. His e-mail is steve.h.gunther@intel.com

Frank Binns obtained a B.S. degree in electrical engineering from Salford University, England. He joined Intel in 1984 after holding research-engineering positions with Marconi Research Laboratories and the Diamond Trading Company Research Laboratory both of the U.K. Frank has spent the last 16 years with Intel, initially holding technical management positions in the Development Tool, Multibus Systems and PC Systems divisions. Frank's last eight years have been spent in the Desktop Processor Group in Technical Marketing and Processor Architecture roles. His e-mail is frank.binns@intel.com

Douglas M. Carmean is a principal architect with Intel's Desktop Products Group in Oregon. Doug was one of the key architects, responsible for definition of the Intel® Pentium® 4 processor. He has been with Intel for 12 years, working on IA-32 processors from the 80486 through the generation beyond the Intel Pentium 4 processor. Prior to joining Intel, Doug worked at ROSS Technology, Sun Microsystems, Cypress Semiconductor and Lattice Semiconductor. Doug enjoys fast cars and scary, Italian motorcycles. His e-mail address is douglas.m.carmean@intel.com

Jonathan C. Hall graduated from Pennsylvania State University with a B.S. degree in Computer Engineering in 1993. He completed his Master's degree in computer science at Rice University in 1996. For the last four years he has worked for the IA-32 architecture and performance team in the Desktop Processor Group at Intel. Jonathan is interested in processor power, performance, and compilation. His e-mail is jonathan.c.hall@intel.com

Interconnect and Noise Immunity Design for the Pentium® 4 Processor

Rajesh Kumar, Desktop Platforms Group-Circuit Technology, Intel Corp.

Index words interconnect, coupling, noise, inductance, domino, cell library

ABSTRACT

For high-performance chip design in deep submicron technology, interconnect delay and circuit noise immunity have become design metrics of comparable importance to speed, area, and power. Interconnect coupling has increased dramatically due to higher metal aspect ratios with process shrinks. Reduction of transistor lengths and thresholds has led to a drastic increase in subthreshold leakage. The Pentium® 4 processor is Intel's fastest processor so far. It contains aggressive domino pipelines, pulsed circuits, and novel circuit families that attain very high speed at the cost of reduced-noise margins. Controlling interconnect RC delay is of paramount importance at such high frequencies. At the same time, the need for a high-volume ramp in the desktop segment necessitates high-density wiring constraints that prevent us from spacing or shielding all critical wires to manage coupling noise. All of these made the task of interconnect and noise design and verification quite challenging.

This paper describes the key innovations and learning in methodology and CAD tools. We first describe our approach to the interconnect high-frequency design problem and our silicon results. We then describe a new proprietary noise simulator (NoisePad) and our noise robust cell library, both of which enabled detailed noise design and analysis for the first time in industry and were critical to our success. Finally, inductance is a major design problem at these high speeds. Our use of a distributed power grid to manage this problem is described.

INTERCONNECT DELAY AND CROSSCAPACITANCE SCALING

With traditional process scaling, interconnect delays have not kept pace with the speedup obtained in transistors. The problem has become significant enough to require entire architectural pipe stages in the Pentium 4 processor for interconnect communication. At the circuit level, widespread use of repeaters has become necessary. To avoid degrading interconnect resistance, the vertical dimension of metals has scaled very weakly compared to the horizontal dimension, leading to extremely high height/width aspect ratios (2-2.2). See Figure 1.

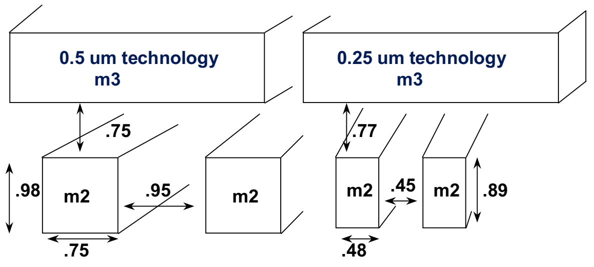


Figure 1: Wire aspect ratio scaling with technology

Nowadays, most of the wire capacitance is to parallel neighboring wires in the same layer (Figure 2), which can get routed together for long distances. This can either lead to a large increase in delay, coupling noise, or min delay problems, depending on the switching direction of neighboring wires. As can be seen from Figure 2, avoiding these delay and noise problems would involve drastically increased wire spacing or extensive shielding. Further, studies on both the Pentium® III and Pentium 4 processor floor plans have clearly shown that we tend to be interconnect limited for die area, which increases the penalty for spacing and shielding.

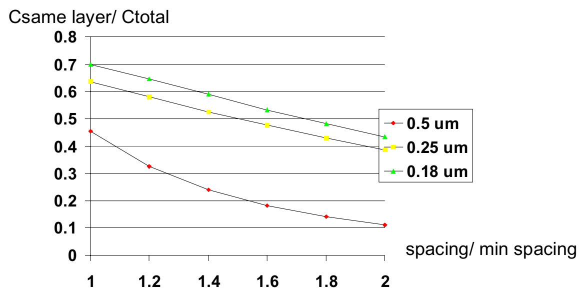


Figure 2: Coupling capacitance scaling with technology

Thus, there is a fundamental design tradeoff between a simple, robust, wiring solution employing extensive spacing and shielding vs. an aggressive solution employing short wiring with only judicious shielding leading to high density. The latter requires sophisticated CAD tools, has more risks, but ultimately is much more optimal for a high-volume product. It was therefore the choice for the Intel® Pentium 4 processor.

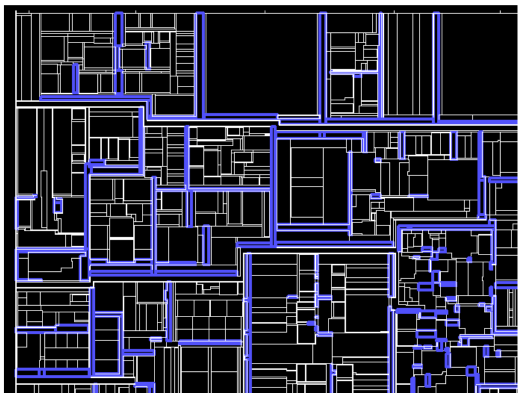


Figure 3: Dedicated repeater banks in the Pentium® 4 processor effectively form a virtual repeater grid

WIRE AND REPEATER DESIGN METHODOLOGY FOR THE PENTIUM® 4 PROCESSOR

Delay, noise, slope limits, and gate oxide wearout were all considered when drafting the guidelines for the wire and repeater methodology. Notable features were an increased emphasis on noise robustness and "pushed process" considerations for delay (repeater distance guidelines were made shorter than optimal for delay with the existing process, in anticipation of end-of-life process trending when transistors speed up a lot compared to wires). Repeater sizing, rather than best delay optimization for non-coupled wires, was picked to be optimal for noise rejection, for equal rise and fall delays, and for better delay in the presence of coupling. Stringent limitations were put on maximum sizing of repeaters, especially in buses, to reduce power supply collapse caused by a simultaneously switching bank of repeaters. The methodology and tools allowed us to use both inverting and non-inverting repeaters. Simple length-based design rules were provided for repeaters, and further optimization was possible through internally developed proprietary tools: NoisePad, ROSES, and Visualizer (net routing and timing) analysis.

The extensive use of dedicated repeater blocks is evident in the Pentium 4 processor floorplan (with repeater blocks highlighted) shown in Figure 3. Further, the net length comparison in Figure 4 shows that although the Pentium 4 processor is a much larger chip, there are very few long nets in it compared to previous-generation chips such as the Pentium III processor. This is even more notable given that the Pentium 4 processor has more than twice as many full-chip nets as the Pentium III processor and has architecturally bigger blocks. If we compare the M5 wire segments of the Pentium III, and Pentium 4 processors, we note that 90% of the M5 wire segments of the Pentium 4 processor are shorter than 2000 microns while the same percentage of Pentium III processor wires are 3500 microns long. These short wires are a key to enabling high-frequency operation.

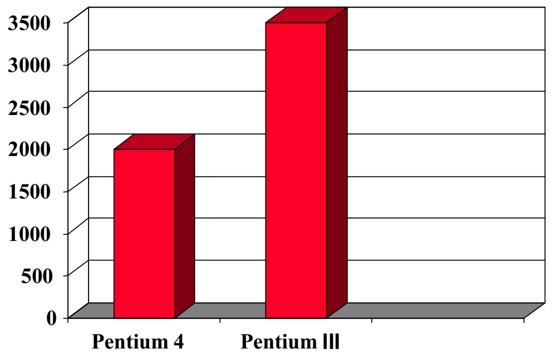
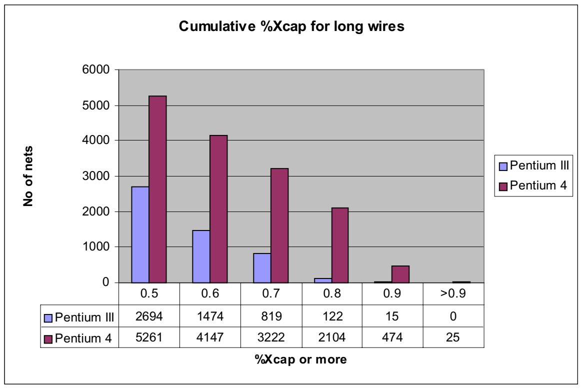


Figure 4: M5 length comparison of global wires for different processors using the same 0.18 um technology

Crosscapacitance and Density Comparative Results of the Pentium 4 Processor Interconnect

The Pentium 4 processor designers' wiring philosophy was to allow short, tight wires. High crosscapacitance was tolerated as the price that had to be paid for dense wiring. Tolerating high crosscapacitance is necessary especially in congested areas of the chip to avoid die growth.



	0.5	0.6	0.7	0.8	0.9	>0.9
Pentium III	2694	1474	819	122	15	0
Pentium 4	5261	4147	3222	2104	474	25

Figure 5: Coupling comparison of Pentium® 4 processor/Pentium® III processor wires

Figure 5 clearly shows that the Pentium 4 processor has significantly more wires with high crosscapacitance than does the Pentium III processor. This aggressive wiring makes additional accuracy in noise CAD tools (discussed later) even more important.

NOISE SOURCES AND TECHNOLOGY TRENDS

There exists a fundamental duality between circuit speed and noise robustness in that we can always make circuits faster by tolerating smaller noise margins. Before looking at this issue specifically from the perspective of the Pentium 4 processor, let us look at noise sources and their scaling.

Interconnect Crosscapacitance noise refers to charge injected in quiet wires by neighboring switching wires through the capacitance between them (crosscapacitance). This is perceived to be the most significant source of noise in current processes (see Figure 6). It is intimately tied to interconnect design for delay and was discussed in the previous section. Device scaling is making the problem worse due to near-end vs. far-end noise effects on resistive metal lines.

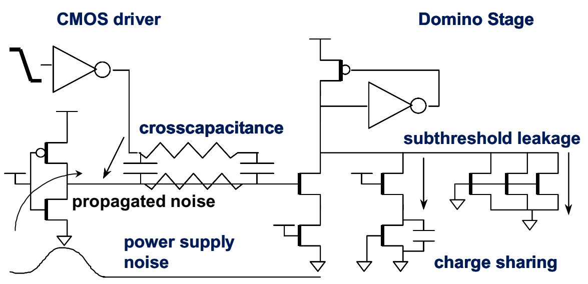


Figure 6: Various noise sources for digital circuits

Charge Sharing Noise is caused by charge redistribution between a weakly held evaluation node and intermediate nodes in a logic stack. This primarily impacts domino nodes, weakly driven pass gate latches, and dynamic latches. The primary technology variable here is the ratio of junction capacitance to gate and interconnect capacitance. For most circuits, this noise is not getting significantly worse with new technology generations.

Charge Leakage Noise in our current processes is mainly composed of subthreshold conduction in nominally off transistors. This current can either charge/discharge a dynamic node or cause the stable state of a weakly held node to be significantly different from rails. This is mainly a concern for wide domino NOR, PLA, and memory arrays. This current increases exponentially with decreasing thresholds and is becoming very significant from 0.18um onwards.

Power Supply Noise is the difference between the local voltage references of the driver and receiver, which can appear as a spurious signal to the receiver and cause circuit failure. It has both low-frequency and high-frequency components. The low-frequency component (IR drop) is managed well by flip-chip C4 packaging, which provides a very low resistance current path. For high-speed transients, the large inductance of the package return causes significant return current to flow through the on-die power grid. For simultaneous switching of wide busses, the impedances in the signal and current return path can be of comparable magnitude leading to large power supply bounce. Power supply noise is a dominant factor in the design of wide domino circuits and in circuits using contention where the AC logic level is shifted with respect to power supply rails.

Mutual Inductance noise occurs when signal switching causes transient current to flow through the loop formed by the signal wire and current return path, thus creating a changing magnetic field (see Figure 7). This induces a voltage on a quiet line, which is in or near this loop. For several signals in a bus switching simultaneously, these noise sources can be cumulative. Unlike crosscapacitance, which is a short-range phenomenon, mutual inductance can be a long-range phenomenon and hence is worse in the presence of wide busses. Faster switching speeds and wider, synchronous bus structures are making this noise very significant in current technologies.

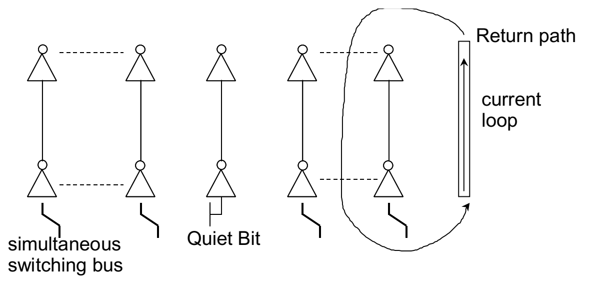


Figure 7: Mutual inductance noise from simultaneous switching on a wide bus

Inductive noise can combine with capacitive noise to cause even worse noise than shown in Figure 7. Because the analysis of inductive effects is highly dependent on layout and is quite complex, the approach is usually to design the problem out through rules rather than analyze arbitrary configurations.

NOISE CHALLENGES ON THE PENTIUM 4 PROCESSOR

The performance goals of the Intel Pentium 4 processor compared to the Pentium III processor were 1.5X–2X higher frequency on the normal (medium) part of the chip and 3X–4X the frequency on the fast (rapid execution engine) part of the chip. These targets require aggressive domino pipelines. In the rapid execution engine, the pipeline is only eight stages deep with the last stage usually feeding the first domino stage after considerable routing. Traditional techniques such as not allowing routing into domino receivers or buffering domino inputs would have added an additional 10-20% latency to the pipe.

Accurate noise analysis using NoisePad and circuit styles such as pseudo-CMOS logic shown in Figure 8 (which provide the logic capability of domino logic and the noise robustness of CMOS) were employed.

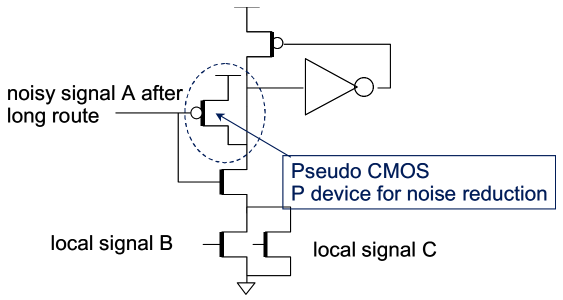


Figure 8: Pseudo CMOS circuit for input noise protection

Pulsed clocking was used in the Pentium 4 processor for lower clock power and load. This made charge sharing protection rather difficult. To reduce power and area, dynamic latches were used extensively as mindelay blockers. These pulsed circuits have no keepers; therefore, increased noise sensitivity and charge leakage had to be verified by noise tools.

A new form of latch called the set-dominant latch was used in the Pentium 4 processor for speed optimization. This weakly held circuit node could get routed into a domino receiver causing increased noise sensitivity.

Process Optimization Consideration for Noise and Leakage

Most design rules and circuit decisions for the Pentium 4 processor, were based on early 0.18 um process specs. We wanted a robust part, which could be pushed for speed later. We expected that the transistor length and leakage targets would be aggressively pushed in our quest for speed in a mature process. Due to these considerations, we employed very large Ioff numbers for our design rules and CAD tools. As shown in Figure 9 by the process trend over time, this was indeed a wise choice: the Pentium 4 processor has scaled well in frequency and still has considerable frequency headroom speed.

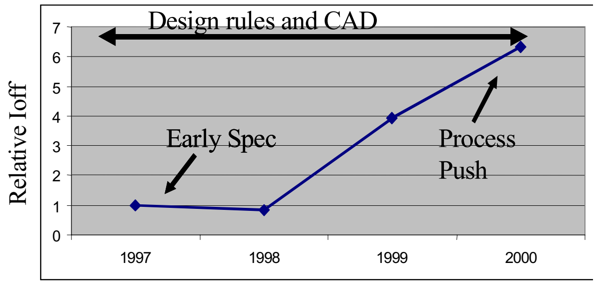


Figure 9: Impact of process push on subthreshold leakage

NOISE ANALYSIS ALGORITHMS

Some amount of noise is unavoidable in digital circuits. The question is deciding when it causes functional failure.

Strongly held static nodes recover after a noise transient and usually incur only a frequency slowdown. Dynamic latches and domino nodes, however, show true functional failure. The node goes to the wrong logic state and may not recover even after the frequency has been slowed down. Latches and other circuits with feedback show a similar failure mechanism.

Small Signal Unity Gain

Prior to our work on the Pentium 4 processor, traditional analysis of noise margins relied on the small signal unity gain failure criteria.

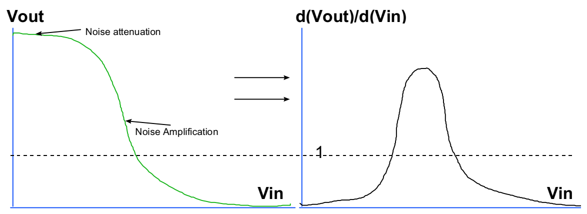


Figure 10: DC transfer function of an inverter illustrating small signal unity gain

As illustrated in Figure 10, for a small change in input noise to a circuit biased at an operating point, the resultant change in output noise is measured. If |d (Output)/ d (Input)| > 1 then the circuit is considered unstable.

Unity gain is a good design metric but is neither necessary nor sufficient for noise immunity. Most aggressively designed paths have some noise-sensitive stages interspersed with quiet stages. We need to allow some noise amplification in the sensitive stage knowing that the quiet stages will finally attenuate it.

Failure Criteria: Noise Propagation

As was mentioned in the previous section, failure criteria based on unity gain tend to be extremely conservative in most cases and are still not proven to be conservative in all cases. Alternately, the entire circuit can be broken into circuit stages, across which noise propagation can be tracked. To do this, we perform an AC circuit simulation of each circuit stage, with noise sources injected in worst-case temporal fashion, combined with noise propagated from previous stages, and measure if any circuit stage failed as a result. In this case, noise can be made to propagate across any number of stages, eliminating the need for any unity gain budgeting. Failure is observed at weakly held nodes such as domino nodes and latches, where the node does not recover after sufficient time. This is very similar to path-based static timing analysis, which allows time borrowing. The computational complexity and memory cost of this approach is the main issue. We made significant CAD innovations to reduce the computational complexity of this approach and implemented this for the Pentium 4 processor in the form of a new noise simulator called NoisePad.

Combination of Noise Sources

Traditionally, different noise sources such as charge sharing, coupling, etc., were characterized separately, and individual maximum budgets were allocated for each source. This is rather conservative. A wide D2 domino NOR node, for example, is very sensitive to coupling at its inputs but has no charge sharing. Some ad hoc approaches to combining noise budgets exist, but the desirable solution is to simulate all noise sources together with no accounting for individual budgets. The simplest way to achieve this is linear superposition. The biggest nonlinear effect is the finite threshold of transistors. For example, a combination of ground bounce and coupling at the input of a transistor leads to a much larger transistor current than does an addition of currents resulting from separate ground bounce and coupling. Another nonlinearity is transistor resistance as a function of drain-source voltage. For example, the peak noise in the event of two simultaneous couplers on a line is larger than the sum of these two events, because the couplee driver resistance increases with an increase in noise magnitude. A third nonlinearity is caused by voltage-dependent parasitics. These are important, for example, when combining charge sharing with coupling effects.

Simultaneous Noise on Multiple Inputs

For multifanin circuits we have to consider not only different noise phenomena, but also their simultaneous occurrence on different parallel inputs. Traditionally, the injection of the same noise on all parallel paths was the worst-case scenario. There are several important cases such as register file arrays where this pessimism can be the deciding factor in the feasibility of the circuit. For example, in a multi-ported register file with a segmented bitline, maximum coupling cannot simultaneously occur on multiple word lines on the same port. Some background noise such as power supply noise may still be present on the other inputs.

DC vs. AC Noise Analysis

Some components of noise such as charge leakage and the low-frequency components of power supply noise have time constants much larger than those of most digital circuits. Effectively, these can be treated as DC waveforms. DC analysis and library characterization are relatively straightforward. Further, it is easy to combine noise sources; e.g., two couplers or coupling with charge sharing, with a DC approach as no computationally costly temporal shifting is required. However, noise sources such as interconnect coupling, charge sharing, etc., have pulsewidths of the same order as those required to charge or discharge most circuits. In this case, approximation of the true waveform with its peak amplitude DC produces gross conservatism. Digital circuits work as "low pass filters" for noise due to their finite transistor resistances and load capacitances. In many matched high-speed circuits, this approximation can lead to a 2X difference in tolerable noise levels. In spite of the severe computational overhead, AC waveform analysis is necessary for the design/verification of sensitive high-speed circuits.

NOISE ROBUST CELL LIBRARY DESIGN

Traditionally, our chips have been designed with fixed cell sizes. The ability to drive different loads has been achieved by providing a finite number of different sizes and in some cases of different P/N skew. For the Pentium 4 processor, we found that additional performance, and area and power optimization, were possible by having a stretchable cell library that didn't have the constraints of fixed cell sizes. Noise robustness was an important consideration for sequential and domino cells. A key innovation for noise robustness was the use of stretchable keepers for domino nodes and sequentials. Traditionally, when assembling domino libraries, keepers were designed to keep additional delay within tolerable limits. For the Pentium 4 processor, instead of the size of keepers being hard-coded, each cell had symbolic constraints describing its leakage and noise metric (no. of pull downs, stacking, etc.), along with its delay metric. The default keeper tried to maximize noise immunity while keeping tolerable delay. As an example, wide fanin domino NORs were provided with significantly larger keepers. Similarly, stacked configurations had larger keepers. However, a designer using NoisePad, optimizing for the actual instance-based noise and speed requirements, could easily adjust this keeper strength. This did not involve creating a new custom cell (unlike other chips) and was widely used for noise suppression.

Each cell could be tuned for its noise environment (as needed) and did not have to follow conservative rules. The symbolic constraints also made the task of process conversion trivial instead of significant since the entire library did not have to be redesigned when leakage changed from a 0.18 to a 0.13 um process.

Another key decision made regarding the cell library was forecasting the optimum leakage of future processes. We predicted that leakage would get much higher for optimized 0.18 and 0.13 um technologies and therefore designed the library to combat this increase. Specifically, for the design of wide domino nodes and array and register file structures, we went with segmented bit-line architecture and disallowed circuits with large numbers of parallel pull downs (except PLA waivers). This design rule allowed us to tolerate significantly higher leakage in the process, which is necessary for transistor performance.

Noise CAD Tool Requirements for the Pentium 4 Processor

In the Pentium 4 processor, we treat charge leakage as DC noise. Interconnect coupling, charge sharing, and noise propagation need to be handled with AC waveform analysis. All noise sources are simulated together without linear superposition. The analysis does not assume maximum budgets on individual noise sources. Regarding simultaneous noise on multiple inputs, by default the same noise is applied to all parallel paths. This can be overridden for speed or area critical paths; in which case, transient noise is analyzed on specified paths with background power supply noise on other paths.

The Pentium 4 processor is primarily custom designed with a library of parameterized/stretchable cells. In past methodologies, custom design resulted in a large overhead for noise analysis because of required characterization. In the Pentium 4 methodology, all cells are treated as custom cells with "on the fly" analysis. This requires no library pre-characterization and thus places no extra overhead on custom design.

NOISEPAD: NOISE CAD TOOLS AND METHODOLOGY

Using the technique of noise propagation, any path can be broken into small circuit stages, which can be analyzed sequentially. Technically, we could perform this analysis with industry-standard SPICE-type simulators. Unfortunately, the throughput available in the Pentium 4 processor design timeframe was not acceptable for either interactive design or batch mode verification. A new transistor-level simulator was developed that allowed a throughput that was orders of magnitude higher than the traditional SPICE approach. The key innovations were symbolic circuit simulation and simplified noise analysis of distributed interconnect.

Symbolic Circuit Template Simulation

To achieve high throughput, the noise simulator reduces/matches circuits to a list of predefined parameterized circuit templates. The differential equations governing these circuit templates have been solved symbolically in a piecewise linear manner and don't need to be solved at runtime. The simulation consists of evaluating these piecewise linear analytical solutions at succeeding time points. Device nonlinearities and voltage-dependent parasitics are dealt with because the model is "piecewise linear" and not just linear. Circuit relaxation is used for DC bias point calculations to handle the DC noise sources. Templates exist for drivers and receivers of CMOS, domino, pass gate, and novel logic types.

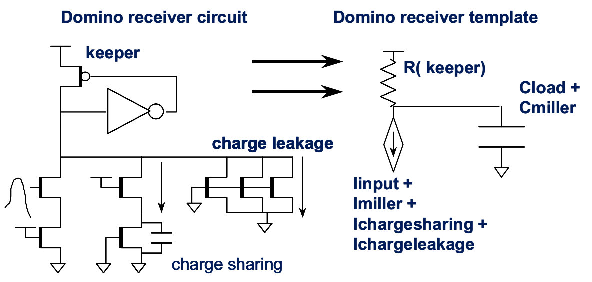


Figure 11: Circuit template idea for a domino receiver

In Figure 11, a piecewise linear waveform of input noise voltage added to the power supply noise creates a piecewise linear current in the receiver. This current is added to other current sources such as charge leakage, charge sharing, and current injected through the gate/drain miller capacitance. The differential equation governing this circuit has a closed form solution, which is known a priori.

Transistor Models

For noise analysis, simple transistor models are often adequate. In this context, some transistors are normally "on", in which case they try to keep a node in its correct logic state, e.g., a domino keeper. These are characterized by a large |VGS| and small |VDS|, meaning they operate in the linear region. Normally, "off" transistors are ones that try to upset the logic state of a node by current conduction. For small or reasonable values of noise, these are characterized by large |VDS| and small |VGS|, meaning they operate in the saturation region. Depending on the gate input noise, these can either be in the subthreshold or strong inversion region. With these simplifications, very computationally inexpensive transistor I-V models were developed and implemented with a precharacterized transistor table look-up model. We used a non-uniform grid to optimize for noise sensitive regions of operation; for example, we used much finer gridding in the subthreshold/weak inversion region.

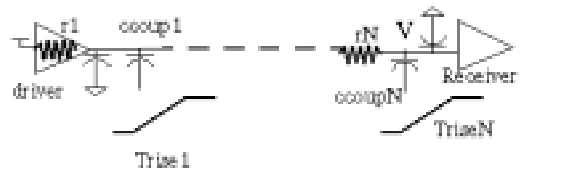
Distributed Interconnect Noise Analysis

The computational complexity of noise analysis is often dominated by the coupling analysis of the distributed interconnect. In the past, interconnect coupling has been dealt with, in a lumped fashion, by putting all coupling capacitance at the end of a line. This produces significant conservatism. Further, for interconnect with side branches, there are no straightforward solutions.

For handling complex interconnect networks, especially from post layout, Asymptotic Waveform Evaluation (AWE) analysis using iRICE has been integrated into our noise simulator.

Elmore Noise Model

To drastically increase the throughput of distributed interconnect noise analysis, a new analytical closed form approximation has been developed for multiple aggressor coupling on a distributed network.



$$tself = \sum_{i=1}^{n}\left(ri \times \sum_{j=i}^{n} ctot_j\right)$$

$$V = VCC\sum_{i=1}^{n}\left(ri \times \sum_{j=i}^{n} \frac{ccoup_j}{trise_j}\right)\left(1 - e^{\frac{-(t + trise_j)}{tself}}\right) \quad t <= 0$$

$$V = VCC\sum_{i=1}^{n}\left(ri \times \sum_{j=i}^{n} \frac{ccoup_j}{trise_j}\right)\left(1 - e^{\frac{-(trise_j)}{tself}}\right) \times e^{\frac{-t}{tself}} \quad t > 0$$

Figure 12: Elmore approximation for noise analysis

This is called the "Elmore model" due to the analogy with Elmore delay used in timing analysis. The idea here is to make the analysis much simpler by reducing the network moments or, in other words, finding the dominant time constant of the network. In Figure 12, *ctotj* is the sum of the total switching and non-switching capacitance on the *jthnode*. All couplers are aligned for worst-case temporal shifts, and they finish switching at time t = 0. NoisePad analysis switches between this simple model and more expensive AWE models, based on heuristics.

FULL-CHIP WIRE NOISE VERIFICATION

The key idea behind the Pentium 4 processor full-chip noise verification is "strobed signaling." A non-restoring node for noise is defined as a node, which if falsely tripped due to noise, will not recover with the passage of time (e.g., domino node or off pass gate latch). A signal is called "strobed," if its logic cone leading to a non-restoring noise node is controlled with a clock (e.g., D1k domino). In this case, the effect of noise on this node may be dependent on clock frequency.

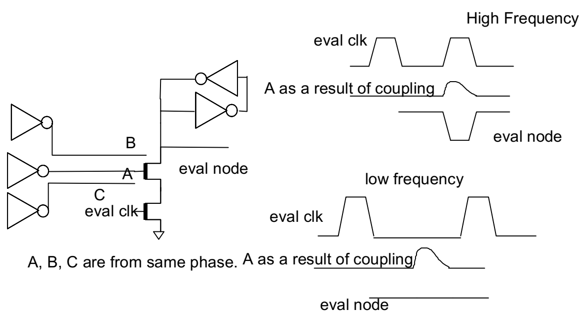
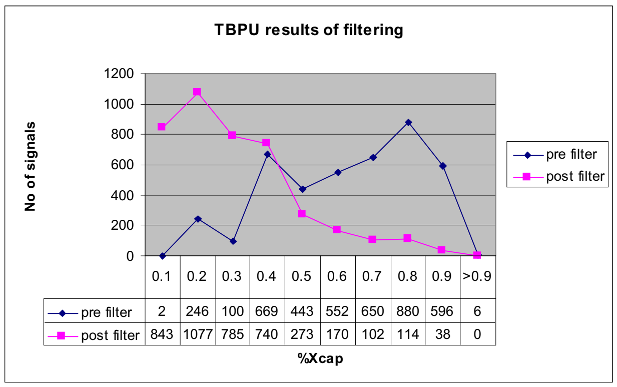


Figure 13: Impact of frequency on noise failure

As shown with the D1-k example in Figure 13, at a lower frequency, the noise will settle down before the signal is sampled and as such will not fail at the lower frequency. In most cases, the timing of aggressors switching for noise is earlier than predicted by max delay timing analysis due to a reduced Miller Coupling Factor (MCF) in the noise case. Further, the worst noise case is usually on fast silicon at high voltage (good for speed). As such, in most cases, we can ignore the cases leading to a slight frequency slowdown in our analysis. The tricky situations are those that lead to excessive frequency slowdown or even worse, frequency shmoo holes. Before spending valuable CAD tool resources on these non-trivial cases, we needed to convince ourselves that the common benign case is indeed the dominant one and therefore the one on which to base our full-chip wiring methodology.

Most full-chip signals are busses (~59,000 out of 72,000 nets), and less than 10% of full-chip signals are "sensitive" (feeding domino receivers or direct pass gate, etc.). Most busses have similar timing among different bits, which should ease the frequency slowdown and shmoo problem. Figure 14 shows the significant effect of this analysis. Most of the effect of this filtering was due to the "required filtering" that characterized frequency slowdown, and very little was due to "valid filtering," which looks for aggressors not switching together.



%Xcap	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	>0.9
pre filter	2	246	100	669	443	552	650	880	596	6
post filter	843	1077	785	740	273	170	102	114	38	0

Figure 14: Impact of frequency independent timing filtering

Frequency Independent Filtering

To solve the rare cases of real noise problems on a strobed signal, we decided to classify noise issues as follows: 1) functional failure at all frequencies; 2) slight slowdown; 3) large slowdown; 4) frequency shmoo hole at a lower frequency as shown in Figure 15; 5) mindelay switching induced noise failure; and 6) excessive coupling causing gate oxide wearout. Issue number 6 was achieved simply through a VCC/2 coupling noise clamp, which was used as a warning. For the rest, we had to implement timing filtering, which understood changing timing relations at different frequencies. Timing filtering was first implemented for the Intel® Pentium® Pro processor as the tool Crosswind [4], and it introduced the concept of valid and required time window filtering; valid window noise 'profiling' or juxtaposition of aggressor noise over the clock period; and rudimentary modeling of drive ratios with fixed thresholds for noise sensitization. Later implementations developed for the Pentium® II and Pentium III processors improved on several aspects of driver and interconnect modeling.

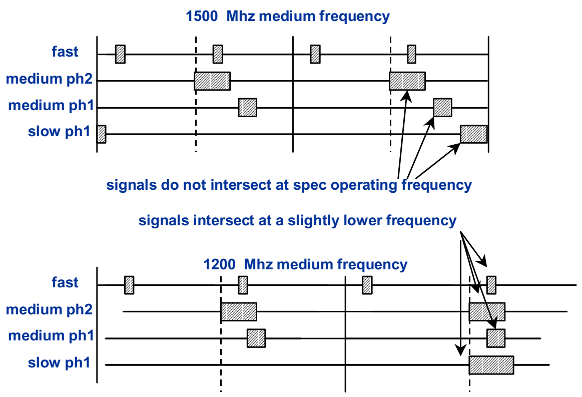

1500 Mhz medium frequency
fast
medium ph2
medium ph1
slow ph1
signals do not intersect at spec operating frequency
signals intersect at a slightly lower frequency
1200 Mhz medium frequency
fast
medium ph2
medium ph1
slow ph1

Figure 15: Frequency shmoo hole

The novel features of timing filtering for the Pentium 4 processor include three modes of frequency analysis (low frequency for burn-in analysis, high frequency for at-frequency noise and delay tests, and all-frequency sweep for noise effects); timing skew between victim and aggressors; required-time filtering with victim recovery; and an interactive graphical waveform interface for timing filter debug.

The design of the Pentium 4 processor brought new challenges to timing filtering because of the complexity of its clocking system. In earlier clocking styles, an excessive slowdown or shmoo hole was usually caused by a very late signal coupling into a signal with early-required time or by the interaction between signals from opposite phases. In the Pentium 4 processor, however, the design incorporates several clocks that are multiples of each other: signals are F(ast), M(edium), and S(low) clocked signals. Not only do signals occur in different phases, but also with different periods. In addition, these differently clocked signals interact as they are not a priori restricted to different regions of the chip. Thus, mid-frequency shmoo holes are much more probable in such a design.

The new approach handles a clocking system with an arbitrary number of phases and an arbitrary number of synchronous clock frequencies by using a *Multi-Frequency Algorithm*.

At very low frequencies, signals activated by different phases are widely separated in time, so much so that they do not interact. This represents the low end of all frequencies to be considered, while the target operating frequency represents the high end. Sweeping frequencies at a small enough increment to catch waveform overlaps is prohibitive due to the complexity of the internal scan. We, therefore, needed a more adaptive algorithm. Here is the entire algorithm with an all-frequency sweep as its outer loop:

For each victim net:

1. Collect aggressor set for a given victim and skew timings appropriately.
2. Map clock edge references onto phases of an appropriate clocking system. For example, a set of aggressors with M and F rising edge references requires a two-phase system.
3. Perform a noise sweep, computing aggressor interaction sets and generating timing “filter table.”
4. Compute the next highest frequency of interaction among signals.
5. Return to step 2 until there is no more interaction among signals.

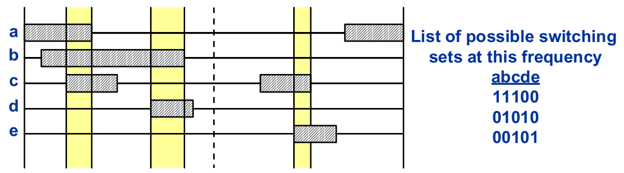


Figure 16: Illustrating logical switching set groups

The most difficult part of the algorithm is to compute the frequencies of interaction, as illustrated in Figure 16. Given that an O(N log N) scan is in the internal loop, the algorithm cannot afford to sweep with a very fine grain to catch all interactions.

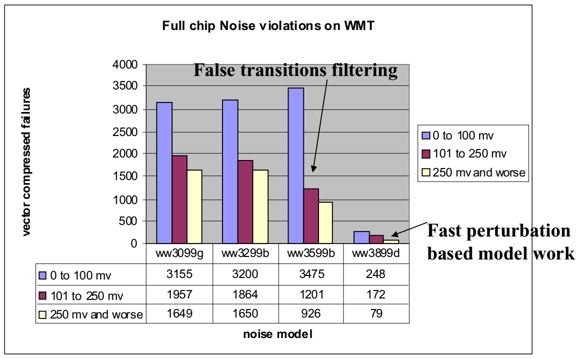
The key to computing the next frequency of interaction is to comprehend the relative velocity of timing edge references as one slows the primary clock. By carefully searching the edges most close to one another and keeping track of their relative velocities, this algorithm can be made reasonably efficient. One difficulty is handling edges that refer to a previous clock phase that are actually moving backward with respect to other timing edges as frequency is increased. To handle this and other difficulties, we developed a general approach to handling both the modular nature of signal timings and measuring the frequency at which they may intersect, based on the concept of relative edge velocity.

Full-Chip Noise Convergence

Detailed noise verification requires a lot of data: circuits, timing information, detailed parasitics, interconnect, etc. For a lead processor like the Pentium 4 processor, “clean” data for all nets are available only very close to tapeout. Further, this detailed model is too slow to turn and, moreover, it is serial in nature. After finding a violation, one has to backtrack through numerous files, models, and schematics to verify if a real problem exists (needle in a haystack scenario). With these incomplete data, trending and schedule predictions are difficult.

To circumvent these problems, simple “perturbation”-based models were built using mathematical spreadsheet software. Parallel probes gather all relevant information about a net (timing, parasitics, length, circuit, etc.) to a total of 87 relevant metrics for each net! Approximately 40 full-chip models were built in one week for various “what if”(perturbation) scenarios. These models looked at tweaking various knobs: number of aggressors, switching probabilities of small aggressors, synchronization of noise propagation with coupling, probability of multiple noise events on same gate, various clock skew assumptions for timing filtering, various frequencies for allowed frequency slowdown, etc., to find reasonable settings and really serious problems but not produce too many false violations. A detailed NoisePad model was used as the starting point for these models. After this analysis, the new noise was assumed to be a slight perturbation around its NoisePad value and predicted by the change in the knob (e.g., changing lumped %xcap from 100% to 50%).

Although these fast models were very crude, they were surprisingly accurate because they did not try to predict the real noise but rather the perturbation (much smaller error). Based on these fast models, another detailed NoisePad model was built with correct knob settings and used for final convergence. As can be clearly seen from Figure 18, this exercise helped us greatly with convergence and saved us an estimated one to two months in our noise convergence schedule. The dramatic decrease in noise violations seen in Figure 17 involved *no* work from the design team!



	ww3099g	ww3299b	ww3599b	ww3899d
🟦 0 to 100 mv	3155	3200	3475	248
🟪 101 to 250 mv	1957	1864	1201	172
🟨 250 mv and worse	1649	1650	926	79

Figure 17: Road to noise convergence on the Pentium® 4 processor

MUTUAL INDUCTANCE METHODOLOGY

At low frequencies, flip-chip C4 packaging provides a very low resistance current return path. For high-speed transients, the large inductance of the package return causes significant return current to flow through the on-die power grid, as shown in Figure 18. For simultaneous switching of wide busses, the impedances in the signal and current return path can be of comparable magnitude leading to large inductive noise.

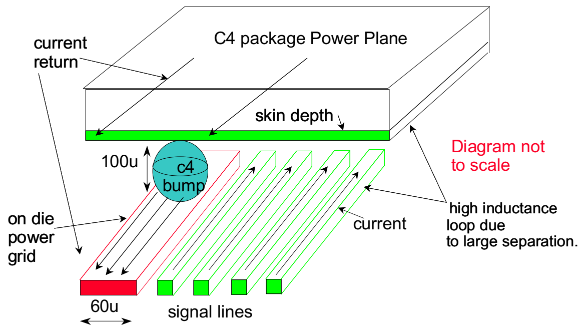


Figure 18: Signal inductance problem with flip-chip packaging

A test chip was fabricated with test structures to measure mutual inductance noise on wide busses. In this chip, signal busses of varying width could be made to switch in any combination, with several combinations of return scenarios, one of which is shown in Figure 19. We were also able to measure simultaneous capacitive and inductive noise, which helped us develop empirical design rules. To keep the area impact small while reducing inductance, a scheme of distributed power supply was chosen for the Pentium 4 processor, where for top-level metals (M6 and M5), a power signal was routed after every 5 signal wires, thus providing a nearby current return and reducing the loop area for inductance. Towards tapeout, a tool for crude inductance estimation was written. This looked for any sensitive circuits (e.g., domino) routed for appreciable distance in the neighborhood and parallel to long, wide busses. By taking the width of the bus, distance from the bus, and length of overlap, an inductance noise metric was used to flag any possible problems. This check was not restricted to wires routed in the same metal layer.

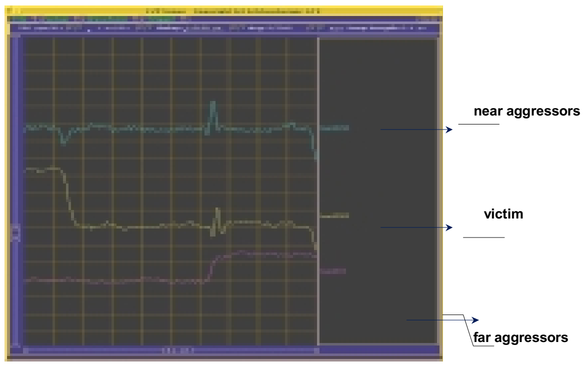


Figure 19: Silicon measurements showing inductive noise

TIMING AND NOISE INTEGRATION

Traditionally, timing analysis (PV) has remained decoupled from noise analysis. As we push both timing and noise limits, there is increasing interaction between the two.

Currently, min delay analysis verifies that all circuits meet their hold time limits while a pulse width/delay check verifies that pulses are wide enough for circuits. In the 0.18 um technology generation, the tool Pathmill[*] is used for min delay analysis. The common algorithm for hold time checks is to ensure the switching data signal does not reach its 50% point before the going away clock reaches its 50% point.

[*]Other brands and names are the property of their respective owners.

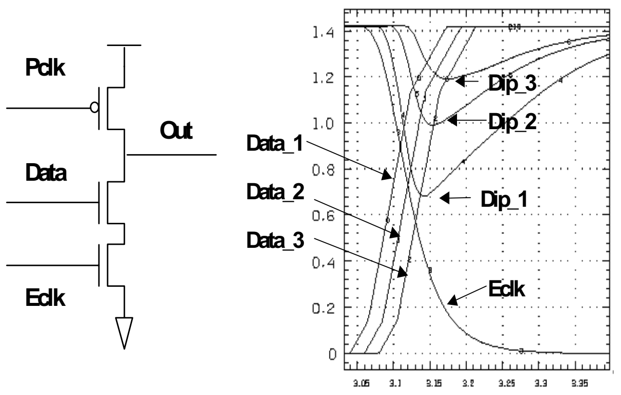


Figure 20: Timing-induced noise

There are some other algorithms, which change the threshold (50% point) or move the check to data output rather than input. These algorithms are inherently flawed because they do not take into account the context-dependent noise robustness of the circuit.

In Figure 20, taking any of the measured values as hold time for a circuit would be completely arbitrary if you didn't know the circuit's noise margin and the other sources of noise that were present. A pulse width/delay checks that the pulse to a circuit is wide enough for it to reach within a certain voltage of a full transition. This check is again arbitrary, without knowing how sensitive that circuit is to incomplete transitions (noise). As an example, we found that a default mindelay Pathmill analysis of the Pentium 4 processor domino library showed several instances where a D1k circuit passing default mindelay (hold checks) would leave a glitch at the domino output that was large enough to cause a complete false transition after the high-skewed static stage. Currently, no design flow would catch these problems, thus causing potential silicon bugs.

Our response was to treat hold checks and pulse width checks as an analog glitch check. The glitch amplitude corresponding to a certain hold time is automatically injected into the noise tools and propagated to succeeding stages to ensure circuit functionality. Thus, we can make tradeoffs between min delay and noise requirements. This new source of noise is combined intelligently and not just added to other traditional sources of noise, such as coupling, taking into account events that are possible logically at the same time. This tradeoff was used quite widely for critical circuits.

Since the design of the Pentium 4 processor, all Intel® timing characterization tools take simultaneous noise margins into account when doing timing analysis for hold, set up, and pulsewidth checks.

SUMMARY

Key findings from the Pentium 4 processor noise and wire design methodologies and CAD tools have been presented. By a combination of aggressive circuit design, short, high-density wiring and noise methodology, and the appropriate CAD tools to help design and verify these, the Intel Pentium 4 processor looks poised to be a successful, fast, reasonably small die product. We have shown that an architecturally larger chip need not lead to longer physical wires if careful methodology and repeater design are used, thus enabling higher frequency. Very aggressive circuit styles have been allowed by innovations in noise CAD tools, which will enable even higher frequencies. High density has been enabled by improved noise methodology, thus allowing aggressive, dense wiring with judicious use of spacing and shielding. The inductance problem, although significant, has been accounted for in the design by our distributed power grid. Circuit styles and a methodology that are robust for leakage will allow us to push the process for speed. Tradeoffs between timing and noise have been enabled by innovations in CAD tools. In general, a lot of care and effort has been put into noise immunity to create a chip that should work robustly in the field Much of this methodology and CAD tool ideas can be incorporated into future chip designs.

ACKNOWLEDGMENTS

We acknowledge all present and past members of our noise group for the all-nighters and Brad Hoyt for discussions. We also acknowledge our DT co-development partners. We acknowledge Paul Madland for guidance and for having a feel for where the silicon problems would really be. And, we acknowledge our management for sticking with our full-chip wire/noise direction, even though at the time, it looked quite risky.

REFERENCES

1. Rajesh Kumar, Eitan Zahavi, Desmond Kirkpatrick, "Accurate design and analysis of Noise Immunity for high-performance circuit design," *Design and Test Technology Conference (DTTC) 1997.* Intel internal document.

2. Eitan Zahavi, Rajesh Kumar et. al., "Novel Methodology and Tools for Noise Immunity Design and Verification," *DTTC 1998*. Intel internal document.

3. Madhu Swarna et. al., "Integrated timing and noise characterization of sequentials for accuracy and increased design space," *DTTC 2000*. Intel internal document.

4. Conley, Kirkpatrick et. al., *DTTC 1995*. Intel internal document.

AUTHOR'S BIOGRAPHY

Rajesh Kumar is currently a Principal Engineer in the Desktop Platforms Group. He received an MSEE degree from the California Institute of Technology (CalTech) and a BTech degree in EE from the Indian Institute of Technology. He joined Intel in 1992 as a designer of the X86 Instruction Decoder of the Pentium Pro processor, working in the areas of microarchitecture, logic, circuit design, and silicon debug. He did the initial research on fundamental circuit limits to high-frequency pipelining, enabling the rapid execution engine for the Pentium® 4 processor. He led the methodology and CAD work for noise, inductance, interconnect, leakage etc., for the Pentium 4 processor. He was the founder and initial chair of Intel's taskforce on crosscapacitance. His current interests are in high-speed/low-power design, parallel/DSP computing architectures, novel non MOSFET devices and conscious computers. His e-mail is rajesh.kumar@intel.com.

Efficient Exploitation of Parallelism on Pentium® Ⅲ and Pentium® 4 Processor-Based Systems

Aart Bik, Microcomputer Software Laboratories, Intel Corp.
Milind Girkar, Microcomputer Software Laboratories, Intel Corp.
Paul Grey, Microcomputer Software Laboratories, Intel Corp.
Xinmin Tian, Microcomputer Software Laboratories, Intel Corp.

Index words: compiler optimization, parallelization, vectorization, SIMD, multithreading

ABSTRACT

Systems based on the Pentium® Ⅲ and Pentium® 4 processors enable the exploitation of parallelism at a fine- and medium-grained level. Dual- and quad-processor systems, for example, enable the exploitation of medium-grained parallelism by using multithreaded code that takes advantage of multiple control and arithmetic logic units. Streaming Single-Instruction-Multiple-Data (SIMD) extensions, on the other hand, enable the exploitation of fine-grained SIMD parallelism by vectorizing loops that perform a single operation on multiple elements in a data set. This paper provides a high-level overview of the automatic parallelization and vectorization methods used by the Intel® C++/Fortran compiler developed at the Microcomputer Software Labs.

INTRODUCTION

The Pentium Ⅲ and Pentium 4 processors are designed to boost application performance and to provide performance scalability. The rich features of the Intel® microprocessors, such as the streaming SIMD extensions [9,10], enable compilers to exploit fine-grained parallelism by vectorizing loops that perform a single operation on multiple elements in a data set. The performance of the majority of scientific, engineering, and multimedia applications with characteristics such as inherent parallelism, a data independent control flow, regular and re-occurring memory access patterns, and localized re-occurring operations performed on the data can be improved by taking advantage of the streaming SIMD extensions. Dual- and quad-processor systems based on the 32-bit Intel® architecture provide opportunities for the compiler to exploit medium-grained parallelism by generating multithreaded code that uses multiple control and arithmetic logic units.

In this paper, we present the high-level software architecture of the automatic parallelization and vectorization methods used by the Intel C++/Fortran compiler developed at the Microcomputer Software Labs. We describe the static and dynamic analysis technologies implemented to enable the efficient generation of parallel code. We follow this with a description of multithreaded and vector code generation. A number of optimization technologies, such as alignment optimizations, advanced instruction selection, multi-entry threading technique, and Profile-Guided-Optimization (PGO) of parallel code, are also presented. We also discuss the results of experiments with automatic vectorization and parallelization on systems based on the Pentium Ⅲ and Pentium 4 processors.

COMPILER ARCHITECTURE OVERVIEW

The approach taken by the Intel C++/Fortran compiler to exploit implicit parallelism in serial code is organized into three stages: program analysis, program restructuring, and parallel code generation.

Program Analysis

Program analysis performs a control flow, data flow, and data dependence analysis [1,3,4,11,12] to provide the compiler with useful information on where implicit parallelism in the input program can be exploited.

The data dependence analyzer is organized as a series of tests, progressively increasing in accuracy as well as time and space costs. First, the compiler tries to prove independence between memory references by means of simple, inexpensive tests. If the simple tests fail, more expensive tests are used.

Eventually, the compiler resorts to solving the data dependence problem as an integer linear programming problem that is attacked by the powerful but potentially expensive Fourier-Motzkin elimination method [7].

Program Restructuring

Program restructuring focuses on converting the input program into a form that is more amenable to parallelization. For example, if static data dependence analysis of a program fails to prove independence, then the Intel C++/Fortran compiler has the ability to generate dynamic data dependence tests to increase the opportunities for exploiting implicit parallelism in a program. An example of this is given below.

```
void init(char *p, char *q) {
    int i;
    for (i = 0; i <= 255; i++) p[i] = q[i];
}
```

Without any further information, the compiler must conservatively assume that the two pointers could refer to overlapping regions in memory. Conversion into multi-version code, however, yields a fully data-independent loop in the true branch that can be optimized accordingly.

```
void init(char *p, char *q) {
  int i;
  if (p+255 < q || p > q+255)
    for (i = 0; i <= 255; i++) p[i] = q[i]; /* dependence free */
  else
    for (i = 0; i <= 255; i++) p[i] = q[i];
}
```

Other examples of transformations that are done during program restructuring are traditional compiler optimizations (such as constant/copy propagation and constant folding [1,3]), loop transformations (such as loop interchanging or loop distribution [11,12]), and idiom recognition (such as the detection of reductions or other operations). An example of the latter category is shown below, where converting an if-statement into a "MAX"-operator makes the loop more amenable for analysis and, eventually, parallelization.

```
for (i = 0; i < N; i++) {                  for (i = 0; i < N; i++) {
   if (a[i] > x)  x = a[i];        ◊          x = MAX(a[i], x);
}                                          }
```

Parallel Code Generation

Finally, parallel code generation consists of converting serial code into semantically equivalent multithreaded code or SIMD instructions. Both these conversions are outlined in the next sections. An in-depth presentation of vectorization is given in [5].

AUTOMATIC PARALLELIZATION

Automatic parallelization is a promising technique that can take advantage of shared-memory multiprocessors based on the Pentium III and Pentium 4 processors.

These systems can potentially deliver near supercomputer performance to mainstream computing. On a multiprocessor system, however, parallelizing inner loops usually does not provide sufficient granularity of parallelism. Thus, our focus for automatic parallelization is to exploit medium-grained parallelism to utilize a multiprocessor effectively. In this section, we describe the parallelization methods used by the Intel C++/Fortran compiler for automatic multithreaded code generation.

Finding Parallel Loops

Finding effective parallelism is one of the critical steps in generating efficient multithreaded code [6,8,11,12]. Based on the control flow graph, the data flow graph and the symbol table, the loop analyzer takes the following steps:

- Finds all loops within the serial code and builds a loop hierarchy structure. It fills up loop parameters such as trip count, lower bound, upper bound, and pre-header.
- Performs data dependence analysis to classify loops. Loops without loop-carried data dependencies are marked as loops that can be made parallel.
- Performs static or dynamic granularity estimation for each loop that can be made parallel. Multithreaded code for a parallel loop will be generated if and only if parallelization of the loop is profitable.

An example of the optimization is shown below.

```
for (k=0; k < 1000; k++) {
     x[k] = k;
     w    = x[k];
     y[k] = w + x[k];
}
```

Parallel loop detection marks this loop as follows.

```
parallel for (k=0; k < 1000; k++) {
      private (k, w), shared (x, y)
      x[k] = k;
      w    = x[k];
      y[k] = w + x[k];
}
```

In this example, the loop is marked as a loop that can be made parallel, and the variables "k" and "w" are marked as **private**. The arrays "x" and "y" are marked as **shared**. In the next section, we discuss variable classification based on liveness analysis.

Variable Classification

Liveness analysis [1,3] is well known and used in many optimizations and transformations. We use liveness analysis to classify the variables in the lexical extent of a loop that can be made parallel.

The **private**, **firstprivate**, and **lastprivate** attributes of variables direct the multithreaded code generator to implement privatization accordingly.

The **shared** attribute of a variable tells the multithreaded code generator to generate code that shares the memory location of this variable amongst multiple threads.

The following compilation rules are used to classify all variables referenced in a parallel loop:

1. A variable is marked **private** if and only if it is not live-in and not live-out on the current loop.
2. A variable is marked **firstprivate** if and only if it is live-in and not live-out on the current loop.

3. A variable is marked **lastprivate** if and only if it is live-out and not live-in on the current loop.
4. A variable is marked **shared** if and only if it is live-in and live-out on the current loop.

For the following example, the loop can be made parallel. Liveness analysis yields var-set = {a, b, c, k, n, x}, live-in-set = {a, b, c, n}, and live-out-set = {a, c, x}.

```
int foo(int b, int n, float c[]) {
    int   x = 101, k, a = 10;
    for (k=0, k < n; k++) {
        x = 5;
        c[k] =  x + a – b * k
    }
    return (a + x + c[0]);
}
```

Using compilation rules 1-4, variables "n" and "b" are marked **firstprivate**. Variables "a" and "c" are marked **shared**. Variable "x" is marked **lastprivate**. Variable "k" is a special form of a **private** variable: it is an induction variable. The data-race condition introduced by such variables is removed by induction variable privatization.

Static Granularity Estimation

Parallelizing a loop can result in slower execution if the overhead of dispatching/scheduling threads and sharing resources is significant compared to the total workload performed by the loop. The Intel C++/Fortran compiler handles this by examining all the operations in the loop body, estimating the grain-size per loop iteration on the targeted microarchitecture, and multiplying this by the loop trip count to arrive at an estimate of the total workload of the loop.

For loops with known trip counts, this value is compared, at compile time, to an experimentally determined profitable workload threshold to see if the loop should be multithreaded. Loops with a workload exceeding this profitable workload threshold will normally speed up when executed in parallel threads. For loops with unknown trip counts, the workload is expressed as a function of the trip count, and the compiler generates code to dynamically evaluate this expression to determine whether the loop should be executed with multiple threads.

Note that this solution avoids all dispatching/scheduling overhead and sharing of resources, if multithreaded execution is not profitable.

For the following example, the compiler generates an expression "(upper - lower) * grain-size" to compute the workload at runtime, based on the lower and upper bound and estimated grain-size.

```
void  foo(int lower, int upper) {
        int i;
        for (i=lower; i<upper; i++) {
           /* grain-size (in units of ops) */
        }
}
```

The granularity estimation has the following form.

```
trip_count  = upper - lower;
workload   =  trip_count * grain-size;

if (workload > (profit_probability *
                PROFIT_WORKLOAD_THRESHOLD) {
     /* multithreaded execution of the loop */
}
else {
     /* serial execution of the loop */
}
```

The profitable workload threshold (expressed in units of ops) is a global constant applicable to all loops. The threshold comparison can be modified with a command line option that sets the probability of profitable parallel execution ("profit_probability"). The workload is then compared to the experimentally determined profitable workload threshold multiplied by this probability. The value "0.0" causes the loop to be always executed as a multithreaded loop, whereas the value "1.0" causes multithreading to be used only if the workload exceeds the profitable workload threshold. The user can use any intermediate value to cause multithreaded execution of loops with low workloads that may still benefit from being made parallel.

Profile-Guided Granularity Estimation

Beyond the static granularity estimation, in the PGO mode of our compiler, we have implemented profile-guided granularity estimation to evaluate the workload, based on the execution count of basic blocks and branch probability. It is well known that compilers are often able to generate better code with the knowledge of likely execution paths. It is even more important for a parallelizing compiler to have the knowledge of the most frequently executed regions in a program, in order to determine if generating multithreaded code is profitable or not. Suppose that for the following code sample, we have the train data set "lower = 0" and "upper = 100." The profiler computes a "branch-taken" probability of "0.98" on the true branch and "0.02" on the false branch. The execution count of the loop pre-header (viz "i = lower") is "1", and the execution count of the loop header is "100."

```
void  foo(int lower, int upper) {
        int i;
        for (i=lower; i<upper; i++) {
           if (i>lower+1) {
                /* TRUE-grain-size (in units of ops) */
           }
           else {
                /* FALSE-grain-size (in units of ops) */
           }
        }
}
```

When these gathered execution measurements are fed back into the second pass of PGO compilation, the compiler compares "100 * (TRUE-grain-size * 0.98 + FALSE-grain-size * 0.02)" with the profitable workload threshold at compile time. Multithreaded code will not be generated if the comparison shows that parallelization is not profitable. If, for example, the expression "TRUE-grain-size" is very small, PGO may avoid the slowdown introduced by parallelization.

MULTI-ENTRY THREADING TECHNIQUE

The conventional technology for generating multithreaded code is to generate an independent subroutine for each parallel loop. This is known as the *outlining* technology [6]. In contrast to this conventional technology, we propose a new technology called the *multi-entry threading* technique, which introduces three new concepts in the control flow graph: T-entry (threaded-entry), T-ret (threaded-return), and T-region (threaded-code-block). The ideas behind the new technology are as follows:

- The T-entry node contains the data environment for each thread that is necessary to build communication between the *invoker* (master thread) and the *invokee* (worker thread).
- The T-ret node informs the multithreaded runtime system about termination of the thread.
- A T-region is defined by a [T-entry, T-ret] pair and is kept inlined in the user-defined subroutine.
- Within a *single* user-defined subroutine, *multiple* [T-entry, T-ret] pairs are permitted to represent multiple T-regions.
- The [T-entry, T-ret] pairs can be nested (e.g., [T-entry, [T-entry, T-ret], T-ret]) to represent nested parallelism.

The main feature of the *multi-entry threading* technique is to keep all newly generated T-regions for parallel loops inlined in the same user-defined subroutine, without splitting them into independent subroutines. This technique provides subsequent compiler phases with more potential to optimize the code.

The following is an example of multithreaded code generation using the *multi-entry threading* technique.

```
float z[10000], w[10000];
void  foo(void) {
   int k, m, x[5000], y[5000];
   … …
   for (k=0;  k<5000; k++) {
      x[k] = x[k] + y[k] ;
   }
    for (m=0; m<10000; m++) {
      z[m] = z[m] * w[m];
   }
… …
}
```

There are two parallelizable loops in the subroutine "foo." The variables "k" and "m" are marked as **private** induction variables; the arrays "x", "y", "z", and "w" are marked as **shared**. The resulting multithreaded code is illustrated below. The Intel C++/Fortran compiler has adopted the KAI[*] Guide runtime library for thread creation and management.

```
float z[10000], w[10000];
void  foo(void)
{    int k, m, x[5000], y[5000];
      … …
    __kmpc_fork_call(loc, 2, T-entry(_foo_ploop_0), x, y)
    goto L1:
    T-entry _foo_ploop_0(loc, tid, x[], y[]) {
       lower_k = 0;
       upper_k = 5000;
      __kmpc_for_static_init(loc, tid, s, &lower_k, &upper_k, …);
      for (par_k=lower_k,  par_k<=upper_k; par_k++)  {
              x[par_k] = x[par_k] + y[par_k] ;
      }
     __kmpc_for_static_fini(loc, tid);
     T-ret;
   }
L1:
   __kmpc_fork_call(loc, 0, T-entry(_foo_ploop_1));
   goto L2:
   T-entry _foo_ploop_1(loc, tid) {
      lower_m  = 0;
      upper_m = 10000;
      __kmpc_for_static_init(loc, tid, s, &lower_m, &upper_m, …);
      for (par_m=lower_m; par_m<=upper_m; par_m++)  {
          z[par_m] = z[par_m] * w[par_m];
      }
      __kmpc_for_static_fini(loc, tid);
      T-ret;
   }
 L2:
   … …
}
```

The multithreaded code generator inserts the thread invocation call "__kmpc_fork_call" with the T-entry point and data environment (e.g., line number "loc") for each loop. This call into the KAI runtime library will fork a number of threads that execute the iterations of the loop in parallel.

The serial loops are converted to multithreaded code by localizing the loop lower and upper bound, and by privatizing the induction variable. Finally, multithreading runtime initialization and synchronization code is generated for each T-region defined by a [T-entry, T-ret] pair. The library call "__kmpc_for_static_init" computes the localized loop lower bound, upper bound, and stride for each thread according to a scheduling policy. The library call "__kmpc_for_static_fini" informs the runtime system that the current thread has completed one loop chunk.

Compared with the existing *outlining* technology, there are three advantages to the *multi-entry threading* technique for generating efficient multithreaded code:

- The *multi-entry threading* technique does not create separate compilation units for parallel loops, and the required program transformations are very natural and simple. It reduces the complexity of handling separate routines in the optimizer.
-

[*] Kuck and Associates, Inc., an Intel Corporation.

- All generated T-regions for parallel loops are kept inlined in the same compilation unit. This minimizes the impact on other optimizations such as constant propagation, scalar replacement, loop transformation, common expression elimination, and interprocedural optimization.
- Besides global and file-scope static variables, the memory location of a local shared static variable can be accessed naturally by multiple threads without passing an argument on T-entry, since the generated multithreaded code is kept inlined in the user-defined subroutine.

AUTOMATIC VECTORIZATION

The Pentium III and Pentium 4 processors feature a rich set of SIMD instructions on packed integers and floating-point numbers that can be used to boost the performance of loops that perform a single operation on different elements in a data set.

The Pentium III processor introduced the 128-bit streaming SIMD extensions [10], supporting floating-point operations on 4 single-precision floating-point numbers and some more instructions for the 64-bit integer MMX™ technology. The Pentium 4 processor further extended this support for floating-point operations on two double-precision floating-point numbers and widened the integer MMX technology into 128-bit [9]. Because a single instruction processes multiple data elements in parallel, all these extensions are very useful to utilize SIMD parallelism in numerical and multimedia applications.

The Intel C++/Fortran compiler follows the standard approach to the vectorization of inner loops [2,11,12]. First, statements in a loop are reordered according to a topological sort of the acyclic condensation of the data dependence graph for this loop. Then, statements involved in a data dependence cycle are either recognized as certain idioms that can be vectorized, or distributed out into a loop that will remain serial. Finally, vectorizable loops are translated into SIMD instructions.

Consider as an example the loop shown below.

```
double a[100], b[100], c[100];  /* assume arrays start at
                                   16-byte boundaries */
…
for (i = 0; i < 100; i++) {
    a[i] = b[i] - c[i];
}
```

Since there are no data dependencies in this loop, the Intel C++/Fortran compiler translates this loop into the following SIMD instructions for the Pentium 4 processor. Note that because double elements are eight bytes wide and the vector loop processes two elements in each iteration, the upper bound and stride for the offsets into the arrays are 100x8=800 and 2x8=16, respectively.

```
SUB:
   movapd   xmm0, b[ecx]      ;  load      2 DP FP numbers
   subpd    xmm0, c[ecx]      ;  subtract 2 DP FP numbers
   movapd   a[ecx], xmm0      ;  store     2 DP FP numbers
   add      ecx, 16
   cmp      ecx, 800
   jl       SUB               ;  looping logic
```

For loops with a trip count that cannot be evenly divided by the vector length, a cleanup loop is used to execute any remaining iteration serially. In the PGO mode, a profile-guided estimation of statically unknown trip counts is used to determine whether vectorization is actually worthwhile.

Alignment Optimizations

In the previous example, the aligned data movement instruction "movapd" can be used because the compiler has aligned the first elements of the three arrays at a 16-byte boundary. For unaligned (or unknown) access patterns, the compiler must use unaligned data movement instructions, like "movupd." Because there can be a substantial performance penalty for unaligned data references, the Intel C++/Fortran compiler has at its disposal a variety of *static* and *dynamic* alignment optimizations.

In the loop shown below, for instance, the compiler will statically peel off one iteration to align all access patterns.

```
double a[100], b[100];   /* 16-byte aligned */
…                                  a[1] = b[1] - 1;
for (i = 1; i < 100; i++) {        for (i = 2; i < 100; i++) {
  a[i] = b[i] – 1;            ◊        a[i] = b[i] – 1;
}                                  }
```

For cases where the alignment of data structures cannot be determined at compile time, the compiler uses a dynamic loop peeling alignment strategy in which, at runtime, first a few iterations are executed serially until one or several access patterns become 16-byte aligned.

Consider, for instance, a simple initialization loop.

```
char *p = …;
…
for (i = 0; i < 100; i++)  p[i] = 0;
```

Without any further points-to information for "p", the compiler would have to conservatively assume that the access pattern is unaligned. Dynamically peeling off some iterations based on the starting address of the array, can, nevertheless, enforce aligned references.

```
peel = p & 0x0f;
if (peel != 0) {
   peel = 16 - peel;
   for (i = 0; i < peel; i++) p[i] = 0;
}
/* aligned access pattern */
for (i = peel; i < 100; i++) p[i] = 0;
```

Reductions

Although reductions give rise to data dependence cycles, such idioms can be translated into SIMD instructions that compute partial results in parallel. Consider, for example, the accumulation that occurs in the DDOT kernel.

```
double d = 0.0;
for (i = 0; i < N; i++) {
    d += a[i] * b[i];
}
```

This reduction can be implemented as follows. Note that in this fragment, the size of double elements is accounted for in the effective address computations. As stated before, serial cleanup code is generated after the vector loop to deal with odd values of N.

```
      xorpd      xmm1, xmm1        ;  reset accumulator
DDOT:
      movapd     xmm0, a[ecx*8]     ;  load,
      mulpd      xmm0, b[ecx*8]     ;    multiply,
      addpd      xmm1, xmm0         ;      and accumulate
      add        ecx, 2             ;        2 DP FP numbers
      cmp        ecx, N
      jl         DDOT               ;  looping logic

      movapd     xmm0, xmm1         ;  postlude:
      unpckhpd   xmm0, xmm1         ;   add 2 partial
      addpd      xmm1, xmm0         ;     results into
      movsd      [esp], xmm1        ;          scalar d
```

Other reductions (based on any of the operators "+", "-", "*", "&", "|", "MIN" or "MAX") are handled similarly.

Short Vector Mathematical Library

The Intel C++/Fortran compiler comes with a Short Vector Mathematical Library (SVML), developed at Intel® Nizhny Novgorad Labs in Russia (INNL), that provides efficient software implementations for computing (inverse) trigonometric, (inverse) hyperbolic, exponential, and logarithmic functions on (sub)arrays. This library provides a clean interface to operate on packed floating-point numbers.

The library allows the vectorization of loops that contain any of these mathematical functions. Consider, for example, the following loop.

```
for (i = 0; i < 100; i++) {
    a[i] = sin(b[i]) + c[i];
}
```

Using the SVML allows the compiler to proceed with vectorization of this loop as follows (an implementation that passes arguments and results in the xmm-registers is planned as well).

```
SIN:
      lea        ecx, b[esi]
      lea        eax, [esp+16]
      mov        [esp], ecx         ;  define input address
      mov        [esp+4], eax       ;  define output address
      call       _vmldSin2          ;  call SVML
      movapd     xmm0, [esp+16]     ;  read result
      addpd      xmm0, c[esi]
      movapd     a[esi], xmm0
      add        esi, 16
      cmp        esi, 800
      jl         SIN                ;  looping logic
```

Advanced Instruction Selection

Advanced instruction selection is used to vectorize certain frequently occurring operations that can be efficiently mapped onto the SIMD instructions of the Intel architecture. Consider, for example, the following loop (the suffix letter "u" denotes an unsigned constant).

```
unsigned char x[256];
…
for (i = 0; i < 256; i++)
    x[i] = (x[i] >= 20u) ? x[i] - 20u : 0u;
}
```

The Intel C++/Fortran compiler recognizes the saturation arithmetic done in this code fragment (if the result of the subtraction would be negative, the result is saturated to zero) and converts the serial loop into the following SIMD instructions that operate on 16 unsigned characters in each iteration.

```
      movdqa   xmm0, CONVEC  ;  load <20u,….,20u>
SAT:
      movdqa   xmm1, x[eax]
      psubusb  xmm1, xmm0     ;  perform  16 saturated
      movdqa   x[eax], xmm1   ;      subtractions
      add      eax, 16
      cmp      eax, 256
      jl       SAT            ;  looping logic
```

The compiler also carefully selects the instructions that are used to implement scalar expansions, certain type conversions, and non-unit stride references. In addition, the use of bit-masks supports the vectorization of singly nested conditional statements.

For a detailed presentation of all the vectorization methods used by the Intel C++/Fortran compiler, we must refer to [5].

EXPERIMENTAL RESULTS

In this section, we discuss the results of some experiments with automatic vectorization and parallelization. Consider, for instance, the following code that computes the product of a double-precision floating-point matrix and vector.

```
for (i = 0; i < n; i++) {
  double d = 0.0;
  for (j = 0; j < n; j++) {
     d += a[i][j] * y[j];
  }
  x[i] = d;
 }
```

In the graph shown in Figure 1, we present the speedup (uniprocessor vs. multiprocessor execution time) obtained by automatic parallelization of the outermost loop in this kernel on a dual 500MHz. Pentium III processor for varying matrix orders. In the same figure, we also show the speedup of serial vs. parallel execution obtained on a quad 550MHz. Pentium III processor. Speedups up to 3.2 and 1.6 are obtained for the quad and dual system, respectively.

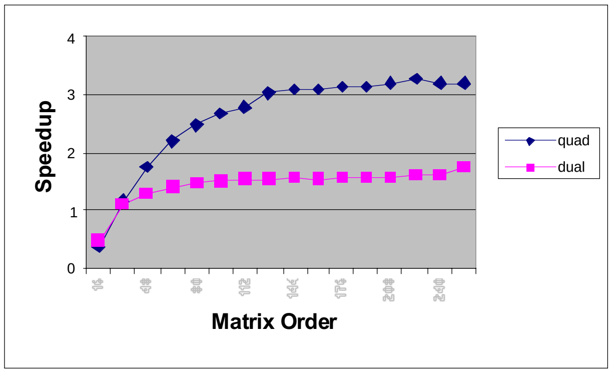


Figure 1: Speedup for matrix x vector on a dual and quad Pentium® III processor

As another example of automatic parallelization, consider LU-factorization without pivoting.

```
for (k = 0; k < n-1; k++) {
  for (i = k+1; i < n; i++) {
      a[i][k] = a[i][k] / a[k][k];
      for (j = k+1; j < n; j++)
          a[i][j] = a[i][j] - a[i][k] * a[k][j];
  }
}
```

In this fragment, loop-carried data dependencies prohibit parallelization of the outermost k-loop. The iterations of the i-loop, on the other hand, can be executed in parallel. In Figure 2, we show the corresponding speedup on a dual and quad shared-memory multiprocessor for varying matrix orders. Despite the fact that the outermost loop has to remain serial, speedups up to 1.3 and 2.6, respectively, are still obtained.

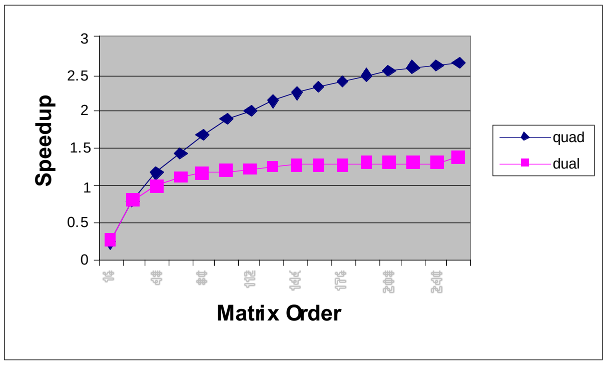


Figure 2: Speedup for LU-factorization on a dual and quad Pentium® III processor

In Figure 3, we show the speedup (serial vs. vector execution time) obtained on a 1.5GHz. Pentium 4 processor by automatic vectorization of a single-precision dot-product kernel (SDOT) and a double-precision dot-product kernel (DDOT) for array lengths ranging from 1 to 64K. For comparison, we also present the speedup obtained by a hand-coded assembly version of the latter kernel (ASM, courtesy Henry Ou). Execution times were obtained by running the kernel many times and dividing the total execution time accordingly, so that for data sizes that fit in the 256KB L1 cache, effectively “warm cache behavior” is measured.

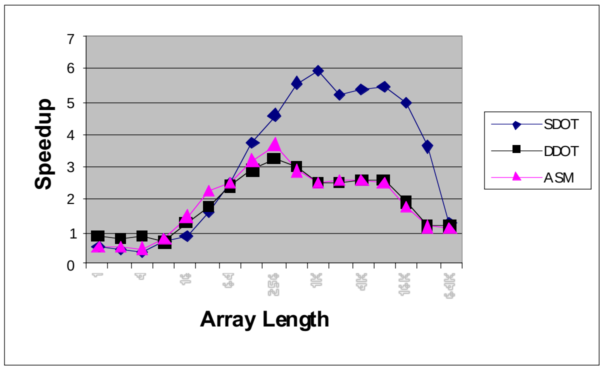


Figure 3: Speedup for dot-product on a Pentium® 4 processor

The performance of the SDOT and DDOT kernels observed after automatic vectorization (counting one floating-point addition and multiplication per iteration) exceeded 3.3 GFLOPS and 1.8 GFLOPS, respectively.

Automatic vectorization of a LINPACK benchmark (available at http://www.netlib.org.benchmark/) boosted the performance of solving a system of linear equations defined by a 100x100 double-precision matrix on a 1.5GHz. Pentium 4 processor from 582 MFLOPS to 700 MFLOPS.

In the last graph shown in Figure 4, we show the speedup obtained on a 1.5GHz. Pentium 4 processor by automatic vectorization of kernels of the form “x[i] = F(y[i])”. The experiments are done for three different double-precision floating-point functions, supported by SVML, and array lengths varying from 1 to 256, with input sets consisting of uniformly distributed values in the range 0 through 2*_.

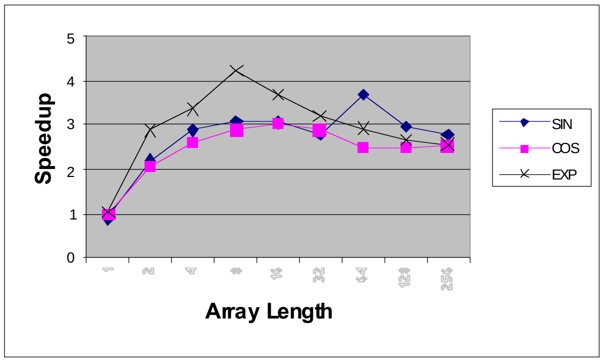


Figure 4: Speedup for math functions on a Pentium® 4 processor

DISCUSSION

The experiments reveal that the automatic detection of implicit parallelism in serial software can provide a very portable way of effectively exploiting SIMD instructions or multiple CPUs on systems that are based on the Pentium III and Pentium 4 processors. Automatic parallelization of the outermost loop in the matrix times vector product starts to speed up for matrices with an order that exceeds 32 on both the dual and quad multiprocessor with an efficiency (Speedup / #processors x 100%) going up to over 80% for larger matrices. Likewise, automatic parallelization of the second outermost loop in an implementation of LU factorization, without pivoting, yields efficiencies of over 60%.

Automatic vectorization of the DDOT kernel yields speedup comparable to the speedup obtained by a hand-optimized assembly implementation. Combining vectorization with efficient software implementations of frequently used mathematical functions already exhibits speedup for arrays with a length of only 2. Another clear advantage of having a vector implementation of mathematical functions is that vectorization of a loop does not have to bail out in the presence of such function calls.

CONCLUSION

Explicitly exploiting parallelism in a program can be a cumbersome and error-prone task. It may require the use of inline assembly to generate the appropriate SIMD instructions or the use of a complicated threading library to take advantage of the computing power available on a multiprocessor. Although such explicit techniques can be extremely effective, they are not portable and greatly complicate program development and maintenance. An alternative approach is to let a compiler do (at least part of) the exploitation of fine- and medium-grained parallelism automatically. With this approach, the compiler analyzes a program that is written in a sequential language for implicit opportunities to exploit parallelism, and it generates code that takes advantage of this implicit parallelism.

In this paper, we provided a high-level overview of the automatic parallelization and vectorization methods used by the Intel C++/Fortran compiler developed at the Microcomputer Software Labs. We have shown that these methods can obtain good speedup on systems based on the Pentium III and Pentium 4 processors, without the need for any source code modifications. Hence, automatically exploiting implicit parallelism provides a convenient way for programmers who are not familiar with the Intel architecture to boost the performance of their applications. In addition, it may even assist expert programmers by minimizing the number of loops that have to be hand optimized to exploit all available parallelism. Finally, the approach allows the automatic parallelization and vectorization of existing serial software, thereby avoiding the potentially huge investments that would be required to hand optimize this code.

More information on Intel's high-performance compilers can be found at

http://developer.intel.com/software/products/

ACKNOWLEDGMENTS

The authors thank the other members of the Proton team for their hard work to implement and test the Intel C++/Fortran compiler. We thank the compiler group at KAI for providing Guide, the multithreading runtime library, and INNL for providing SVML. Both these libraries are currently part of the Intel C++/Fortran compiler.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers—Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Boston, Massachusetts, 1986.

[2] John Randal Allen and Ken Kennedy, "Automatic Translation of Fortran Programs into Vector Form," *ACM Transactions on Programming Languages and Systems: 9:491-542*, 1987.

[3] Andrew W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, Cambridge, UK, 1998.

[4] Utpal Banerjee, *Dependence Analysis*, Kluwer Academic Publishers, Boston, Massachusetts, 1997.

[5] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "An Auto-Vectorizing Compiler for the Intel® Architecture," *Submitted to the ACM Transactions on Programming Languages and Systems*, 2000.

[6] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors," *in Proceedings of CASCON: 76-89, Toronto, ON, November 12-14*, 1996.

[7] George B. Dantzig and B. Curtis, "Fourier-Motzkin Elimination and its Dual," *Journal of Combinatorial Theory: 14:288-297*, 1973.

[8] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. "Detecting Coarse-Grain Parallelism using an Interprocedural Parallelizing Compiler," *in Proceedings of Supercomputing, San Diego, California, December*, 1995.

[9] Intel Corporation, *Intel® Architecture Software Developer's Manual With Preliminary Willamette Architecture Information*, manual available at http://developer.intel.com/.

[10] Shreekant Thakkar and Tom Huff. "Internet Streaming SIMD Extensions," *IEEE Computer: 32:26-34*, 1999.

[11] Michael J. Wolfe, *High Performance Compilers for Parallel Computer*, Addison-Wesley Publishing Company, Redwood City, California, 1996.

[12] Hans Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, NY, 1990.

AUTHORS' BIOGRAPHIES

Aart Bik received his M.Sc. degree in Computer Science from Utrecht University, The Netherlands, in 1992, and his Ph.D. degree from Leiden University, The Netherlands, in 1996. In 1997, he did a post-doc at Indiana University, Bloomington, Indiana, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel Corporation where he is currently working in the vectorization and parallelization group of the Microcomputer Software Labs. His e-mail is aart.bik@intel.com

Milind Girkar received a B.Tech. from the Indian Institute of Technology, Mumbai, an M.Sc. degree from Vanderbilt University, and a Ph.D. degree from the University of Illinois at Urbana-Champaign, all in Computer Science. Currently, he manages the vectorization and parallelization group in Intel's Microcomputer Software Labs. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com

Paul Grey did his B.Sc. degree in Applied Physics at the University of the West Indies and his M.Sc. degree in Computer Engineering at the University of Southern California. Currently he is working in Intel's Microcomputer Software Labs, researching compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., Sun Microsystems, and Silicon Graphics. His research interests include optimizing compilers, advanced microarchitectures and parallel computer systems. His e-mail is paul.grey@intel.com

Xinmin Tian is currently working in the vectorization and parallelization group at Intel's Microcomputer Software Labs where he works on compiler parallelization and optimization. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel, he worked on parallelizing compilers, code generation, and performance optimization at IBM. His e-mail is xinmin.tian@intel.com

•

*Other brands and names are the property of their respective owners.

A Discussion of PC Platform Balance: the Intel® Pentium® 4 Processor-Based Platform Solutions

R. Scott Tetrick, Blaise Fanning, Robert Greiner, Tom Huff, Lance Hacking, David Hill, Srinivas Chennupaty, David Koufaty, Subba Palacharla, Jeff Rabe, Mike Derr
Desktop Platforms Group, Intel Corporation

Index words: Pentium® 4 processor, platform, STREAM, performance, architecture

ABSTRACT

The quest for the balanced PC platform has been with us since the advent of the Personal Computer (PC) in 1982. The basic PC has been dramatically successful, such that a 200-fold increase in computing power has been required. At the same time, the platform has had to improve its capabilities to support these uses, and provide growth for new applications. To restore balance in the platform, processor buses, memory interfaces, and advanced platform capabilities must keep pace with, and even lead, advancements in the processor. This paper describes the technological advances made in the development of the first platform for the Intel® Pentium® 4 processor. After a brief look at PC platform partitioning through the years, the platform partitioning developed for the first Pentium 4 processor platform is discussed, beginning with an understanding of the performance of the processor, and how that is dependent on aspects of the platform. We then present two primary platform improvements for the high-performance PC platform: the 400MHz system bus and the Intel® 850 Memory Controller Hub. After providing an understanding of these platform advancements, we show how these two improvements together complement the Pentium 4 processor computational capabilities by concentrating on results obtained in the standard *SPEC CPU2000*[1] and *STREAM*[2] benchmarks. This combines the high-speed processing of the Intel Pentium 4 processor with platform improvements to provide a dramatic increase in overall performance.

[1]The next-generation industry-standardized CPU-intensive benchmark suite. SPEC designed CPU2000 to provide a comparative measure of compute-intensive performance across the widest practical range of hardware.

[2]A simple synthetic benchmark that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels.

INTRODUCTION

With the first IBM Personal Computer (PC) in 1982, the basic platform architecture of today's PC platform was established. Improvements to this platform were necessitated by improvements to the microprocessor, as dictated by Moore's Law. While some of these improvements were small, others were major, moving also at the pace of Moore's Law. In this paper, we first describe the substantial platform changes that allow us today to support the latest Intel Pentium 4 processor.

In order to understand the impact of these improvements on the Pentium 4 processor platforms, we need to see how these improvements impact the performance of the applications. Previous discussions of the performance of the Pentium 4 processor have described the processor performance as the product of two values, frequency and Instructions Per Clock (IPC). While the increase in frequency is strictly due to advances in microarchitecture and an improved silicon process, the IPC value has a strong platform-level component as well.

In this paper, we highlight two platform advances in the Pentium 4 processor. First, the 400MHz system bus of the Pentium 4 processor, responsible for all data into and out of the processor, is discussed. This high-bandwidth connection provides the necessary throughput for today's performance applications, with headroom for new applications in the coming years. Second, the system bus advanced protocol is discussed. This improved parallelism is best shown by a careful examination of the 82850 Memory Controller Hub. The 82850 is responsible for balancing the bandwidth demands of graphics, I/O, and the processor with two channels of Direct RDRAM.

An obvious question at this point is whether these platform improvements are necessary. We present case studies to show that these platform improvements add to bottom-line performance. This is represented graphically to show how the platform's failure to meet the peak bandwidth demand of the application impacts the overall performance of the microprocessor. For many applications, even average bandwidths can show that the high-performance aspects of the platform of the Pentium 4 processor are required. Finally, we carefully examine the *STREAM* benchmark, which combines the high-performance computational capabilities with the high bandwidth supplied by the platform of Intel's Pentium 4 processor.

THE ARCHITECTURE OF THE FIRST PERSONAL COMPUTERS

In 1982, the first IBM Personal Computer (PC) was introduced. This relatively simple design has held steady for all PC platforms since then. The processor was the Intel® 8088, running at a clock rate of about 5MHz and providing a processor bus connection of about 1.2 megabytes/second. Figure 1 shows a simplified block diagram of the early IBM PC. Moreover, it should be noted that all bandwidth in the system was routed through this processor bus. The bandwidth of this bus provided a ceiling for all the graphics and the I/O in the system.

Contrast this to the 1995 version of the PC architecture, which shows how much the platform architecture changed in just five years. The processor bus had improved 500-fold to over 500 megabytes/second. Moreover, a major platform improvement was made with the addition of the Advanced Graphics Port (AGP) interface. This repartitioning of the platform removed the graphics bandwidth from the I/O interface. This greatly improved the platform capabilities, but complicated the function of the chipset, the 82440BX.

The 2000 PC block diagram shows the platform of the Pentium 4 processor. As you can see, there have been the usual improvements to the interfaces, but a continued repartitioning of the platform. This repartitioning represents a continued improvement in the platform, as a result of advanced speeds and feeds. The growth requirements of the platform are shown in Table 1. We explore the reasons for these platform changes and how they contribute to improved platform performance.

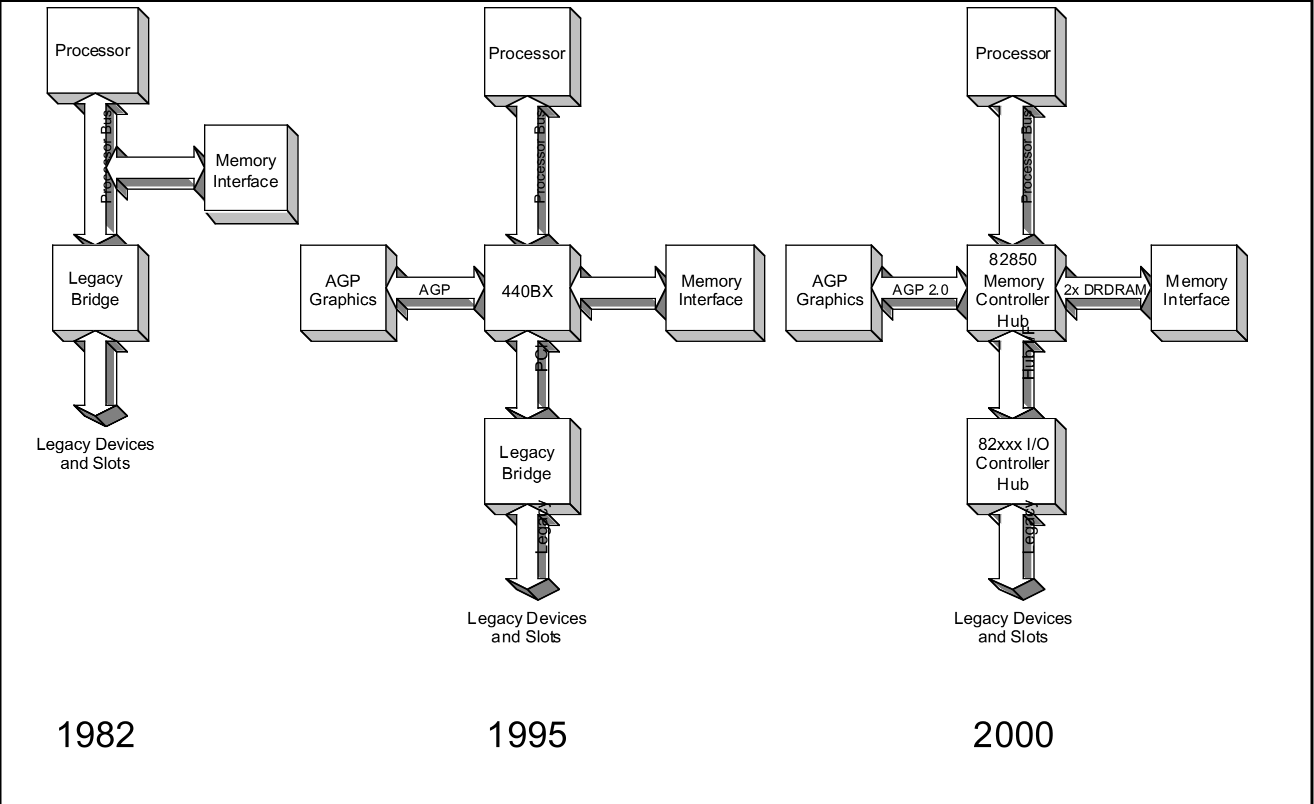


Figure 1: PC architecture over time

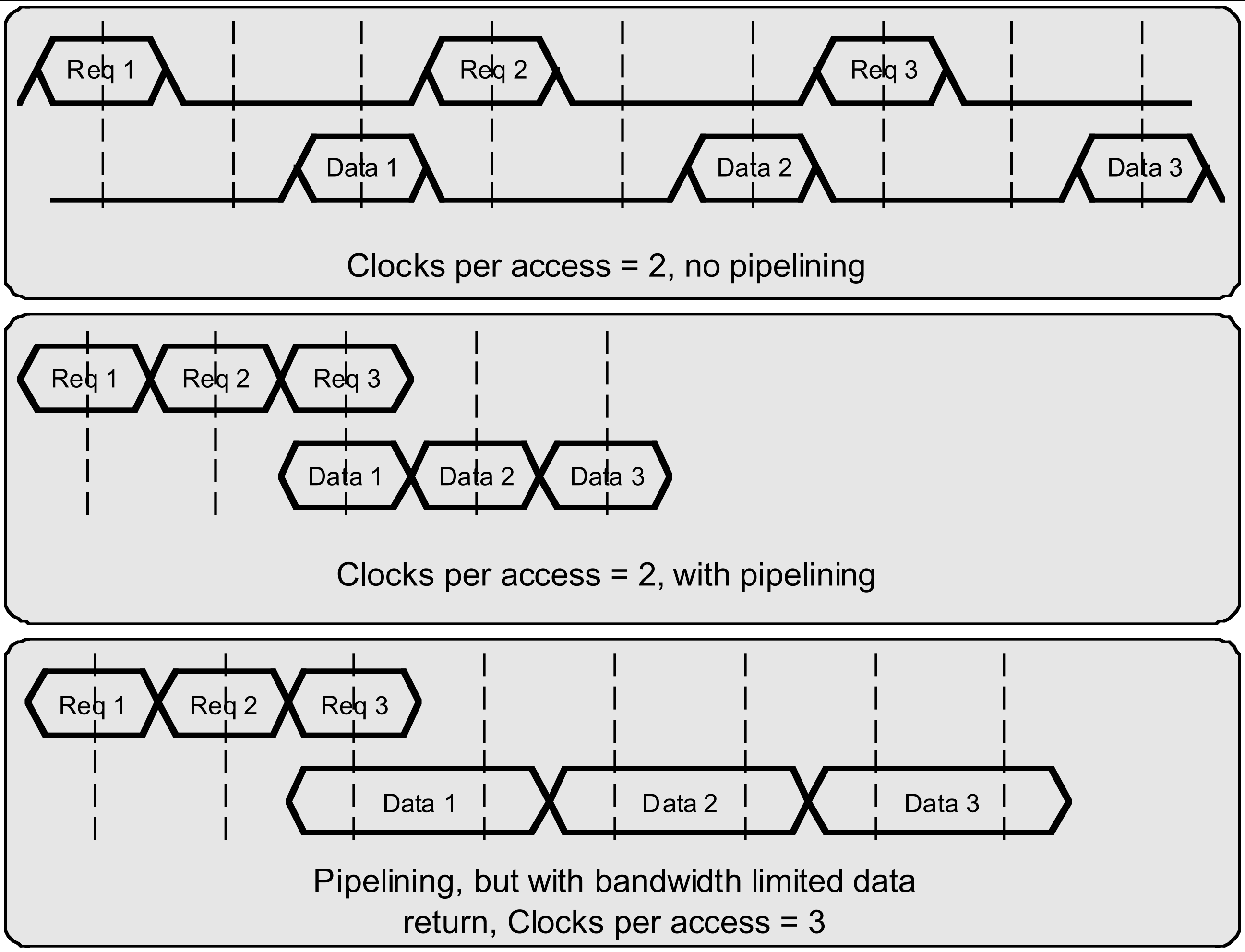

Req 1
Req 2
Req 3
Data 1
Data 2
Data 3
Clocks per access = 2, no pipelining
Req 1
Req 2
Req 3
Data 1
Data 2
Data 3
Clocks per access = 2, with pipelining
Req 1
Req 2
Req 3
Data 1
Data 2
Data 3
Pipelining, but with bandwidth limited data return, Clocks per access = 3

Figure 2: System bus impacts to clocks per access

Platform Interface	1982	1995	2000
Processor Bus Bandwidth	1.2 MB/s	533 MB/s	3200 MB/s
Graphics Bandwidth	N/A	533 MB/s	1066 MB/s
I/O Connection	N/A	133 MB/s	533 MB/s
I/O Bus	1.2 MB/s	Same as I/O Connection	133 MB/s

Table 1: Bandwidth increases over time

How the Platform Contributes to Performance

Before looking at results, let us take a look at the theoretical basis for the impact of the platform on performance. Fundamentally, the time to run an application is

*Execution Time = Instructions * CPI/Frequency*

Where:

Execution Time = application run time
Instructions = number of instructions
CPI = clocks per instruction
Frequency = CPU core frequency

CPI varies with the application, cache hierarchy, and I/O use. It can be further defined as

$$CPI = CPI_{base} + \underset{i}{\ } P_i C_i$$

CPI_{base} is a figure of merit for the application and the core processor microarchitecture. P_i is the probability of event i occurring which has an additional latency in clocks of C_i. C_i values may be very large; for example, if the core clock is 1GHz and the average memory access is 100 ns, C_i is 100 clocks. Unfortunately, C_i for a fixed latency device varies with frequency (doubling our core frequency in the example above also doubles our C_i to 200 clocks). C_i can also be dependent on system state, the ability of the platform to support pipelining, and the queuing impact due to demanded bandwidth mismatches. Examples of these C_i cases are shown in Figure 2.

The examples presented in Figure 2 clearly lead us to the following conclusions, as we want to scale to high frequencies in the processor.

- The connection to the processor must be highly pipelined to improve bus efficiency and avoid queuing requests.
- The bandwidth to the processor must closely match the request rate to queuing responses.

HIGH-PERFORMANCE INTERFACE TO THE PROCESSOR

The *STREAM* benchmark, along with various *SPEC* benchmarks, is used to show how the platform impacts the overall performance of the system.

The *STREAM* benchmark is a simple synthetic benchmark that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. It represents a balance between memory bandwidth and floating-point operations. While the benchmark is artificial, it is similar to a number of applications of interest. Applications with streaming data, such as video editing, format conversion of audio and video, and encryption primitives all have behaviors common to those of the *STREAM* benchmark.

PLATFORM IMPROVEMENTS TO KEEP PACE WITH PROCESSOR

As processor clock speeds continue to grow exponentially, the system memory bandwidth required keeping these processors busy doing useful work grows as well. Ideally, when CPU frequency is doubled, application execution time should be halved. On a given processor architecture with a fixed cache size and caching strategy, the instantaneous system memory bandwidths would double, since the execution engine would still require access to all of the same pieces of memory data. It would, however, require access to them in half of the time required by the slower processor.

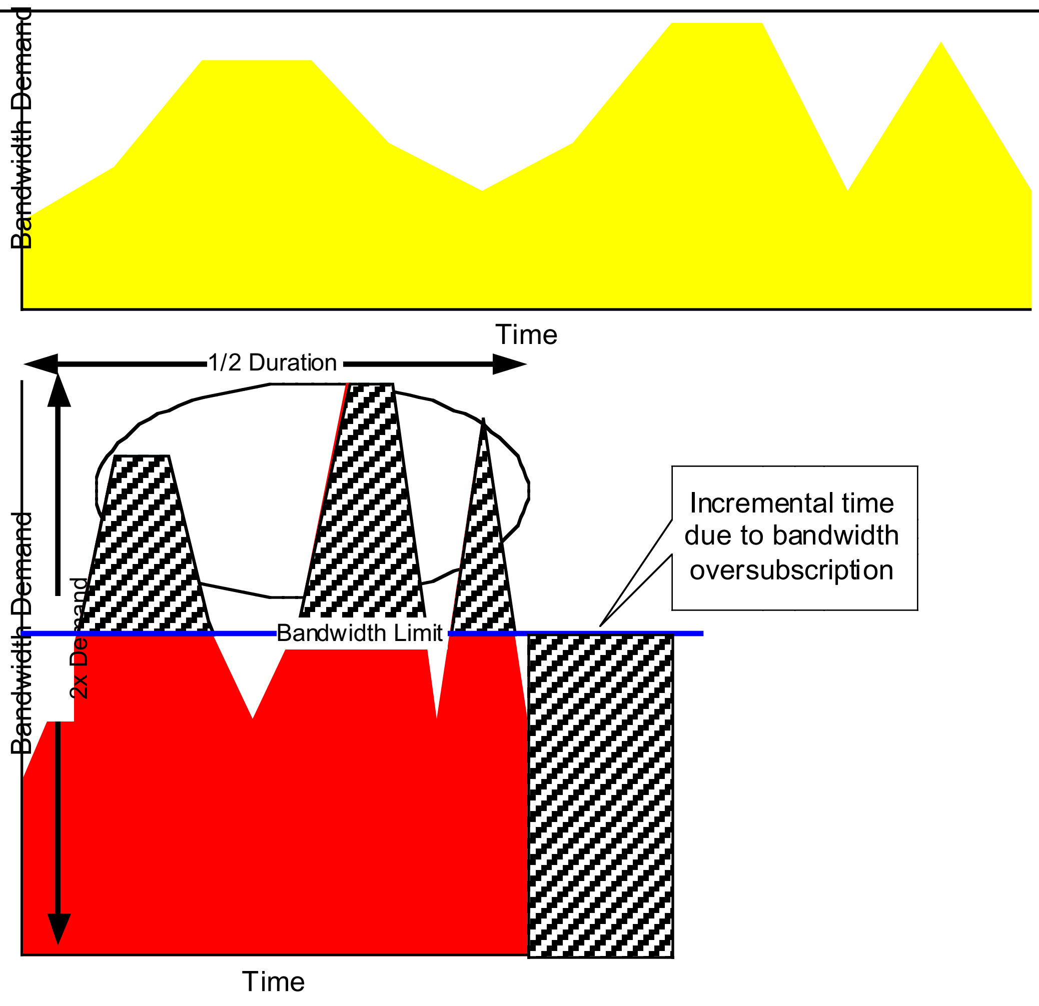


Figure 3: Frequency scaling and memory limitations

System memory application bandwidth vs. application execution time is plotted in Figure 3. The yellow area of the graph indicates the memory bandwidth demanded over time of the applications. If bandwidth is unconstrained, the processor will request memory twice as fast and obtain twice the bandwidth, and scale perfectly. In other words, it completes the application in one-half the time. However, let us assume that the memory bandwidth is constrained as indicated by the dark blue band. The increased memory demand can only be met by an increase in the execution time, as shown in the shaded area. Due to the limited memory bandwidth, the processor that is twice as fast yields less than the 2x speedup due to the finite memory bandwidth. During this process, the area under the bandwidth profile remains constant. Since the application requires its data more quickly, the bandwidth required to satisfy processor requests increases.

It is important to note that this demand profile for memory bandwidth is application specific. Many applications demand memory resources at a fairly low rate, and the platform can accommodate the processor speedup without degradation. Examples of applications that do not demand high bandwidths include word processing and presentation software, such as those components of the SYSmark 2000[3] benchmark.

Systems whose memory and bus implementations limit the amount of bandwidth available to the system execution engines decrease the ability of applications to scale with processor frequency increases. A memory

[3] An application-based benchmark that reflects today's leading-edge software applications for Internet Content Creation and Office Productivity. SYSmark is a registered trademark of The Business Applications Performance Corporation.

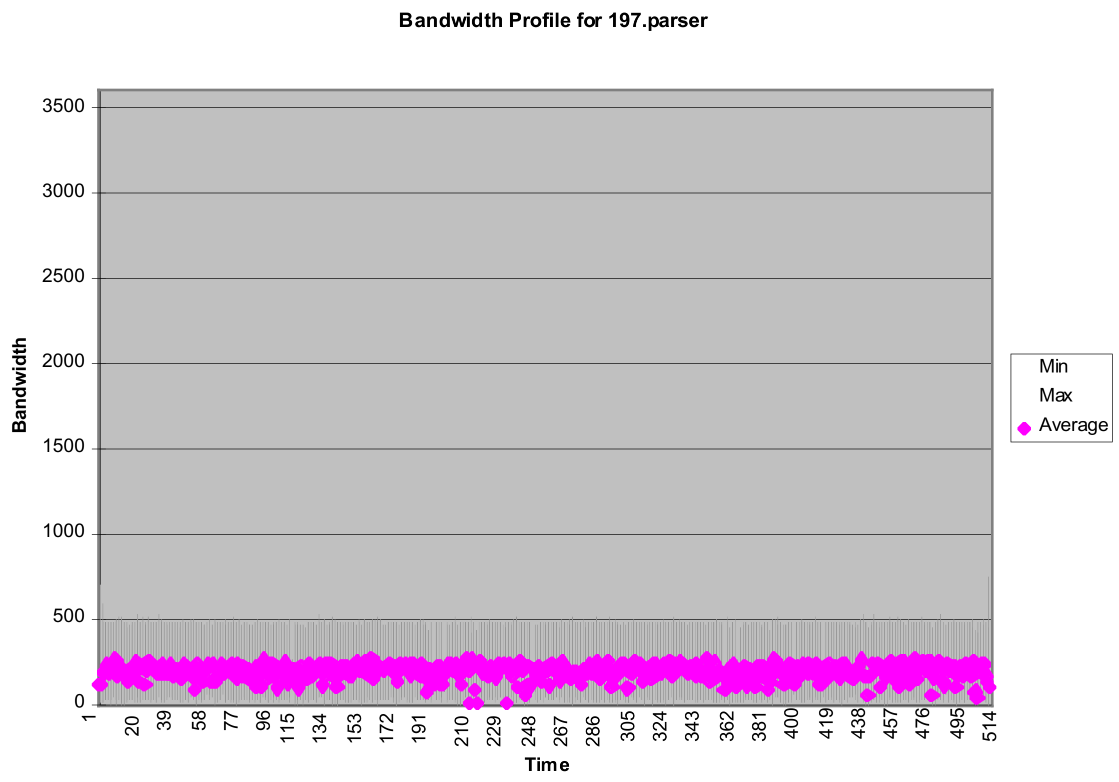


Figure 4: Pentium® III processor 197.parser bandwidth profile

subsystem implemented with a 64-bit PC133 SDRAM or with a 133MHz Pentium® III processor bus would limit the ability of the above example application to scale linearly with processor frequency. Such memory systems and processor buses can ideally deliver only 1.066GB/sec. As a result of this "ceiling", the CPU is forced to wait for some of the pieces of data that it requires during the time the bandwidth limit is exceeded, as indicated by the shaded areas in Figure 3. While the processor is waiting, it is doing no useful work and is not contributing to increased application speed.

In order to analyze the effects on the system of limited and expanded bandwidth, measurements were taken on a system with a Pentium III processor running different components of the *SPEC* CPU benchmark suite. As one might expect, there was wide variation in the amount and profile of memory bandwidth required by the discrete benchmark components. However, the benchmarks could basically be divided into two categories: benchmarks that were already bandwidth limited or nearly bandwidth limited and benchmark components that required very little bandwidth and would scale well even without the bus and system enhancements brought by the Pentium 4 processor, the Intel850, and RDRAM memory.

The first class of benchmarks, those with low-bandwidth requirements in the Pentium III processor generation, included a number of the integer benchmarks, such as the 197.parser and the 168.wupwise. Their execution profiles on a 1GHz platform with a Pentium III processor, using an Intel840 chipset, showed fairly low system bandwidth consumption. The bandwidth results for the 197.parser are illustrated Figure 4. Note that Figure 4 illustrates the average bandwidth consumption in a 1-second slice using a solid dot, while the high- and low-bandwidth limits during that same slice are shown with a gray line.

The bandwidth profile for the 197.parser demonstrates a fairly consistent average bandwidth requirement of around 250MB/sec, over the duration of the benchmark execution time. During this time, the peak data requirements stay below 500MB/sec. As a result, memory and processor bus implementations that limit processor data to around 1GB/sec should not limit even a 2GHz execution engine.

The other class of applications found in the *SPEC* CPU benchmark suite are those that are clearly bandwidth limited on the Pentium III processor platforms. Floating-point applications such as the 179.art, the 171.swim, and the 172.mgrid illustrate the problem most vividly, although integer applications such as the 181.mcf also

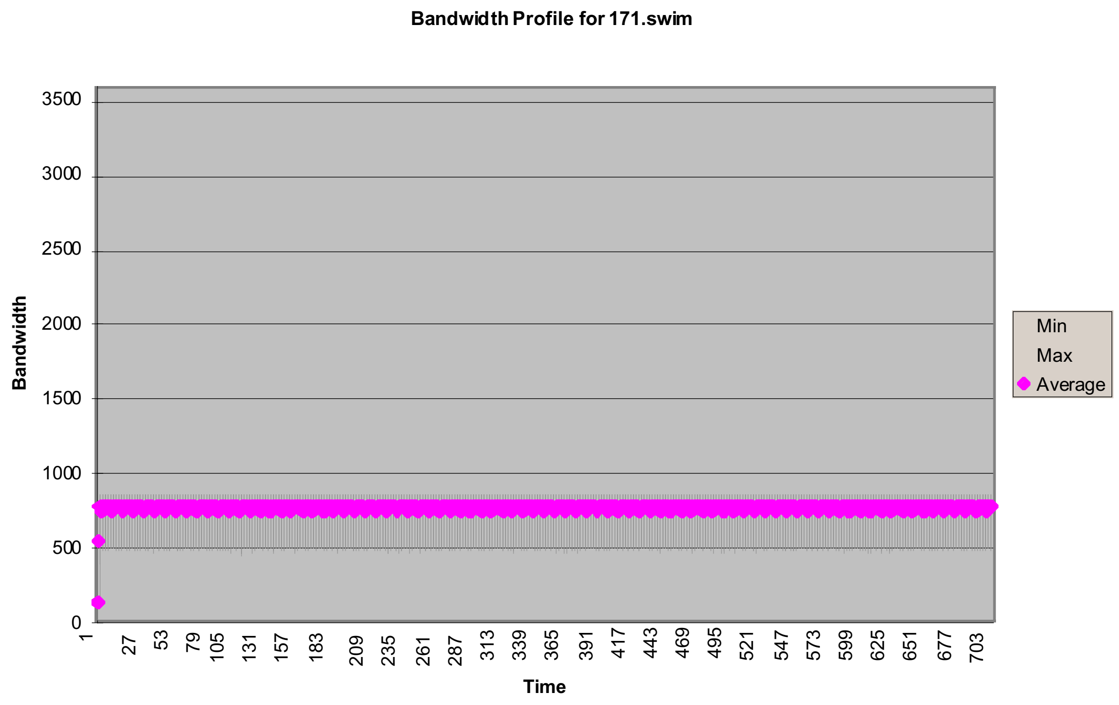


Figure 5: Pentium® III processor bandwidth profile for 171.swim

demonstrate serious bandwidth "hunger." Figure 5 illustrates the bandwidth profile for the 171.swim.

The graph clearly illustrates the bandwidth limitation of the existing Pentium III processor bus, since its average processor bus utilization is around 800MB/sec or 80% of the ideal processor bus bandwidth. If processor frequency scaling is to return benefits on this application, the bus and memory system bandwidth capabilities must clearly be increased. Existing structures do not support application runtime scaling, and applications such as the 171.swim executing on Pentium III processor platforms clearly fall into the class of being "bandwidth-limited."

Given vectors **a, b** and **c**, and scalar q, the *STREAM* benchmark measures the memory bandwidths for the following operations.

Function	Vector Operations
Copy	$\mathbf{a} = \mathbf{b}$
Scale	$\mathbf{a} = q\mathbf{b}$
Sum	$\mathbf{a} = \mathbf{b} + \mathbf{c}$
Triad	$\mathbf{a} = q\mathbf{b} + \mathbf{c}$

The sizes of the arrays are set much larger than processor cache sizes to guarantee memory is exercised. By performing both floating-point and bus operations, platform balance can be assessed. *STREAM* operations are typical for a number of new application classes, where streaming data are required to be delivered to the processor, computed, and delivered to a peripheral at high speed. This new workload is required in such things as speech recognition, video editing, and Internet servers with streaming datatypes.

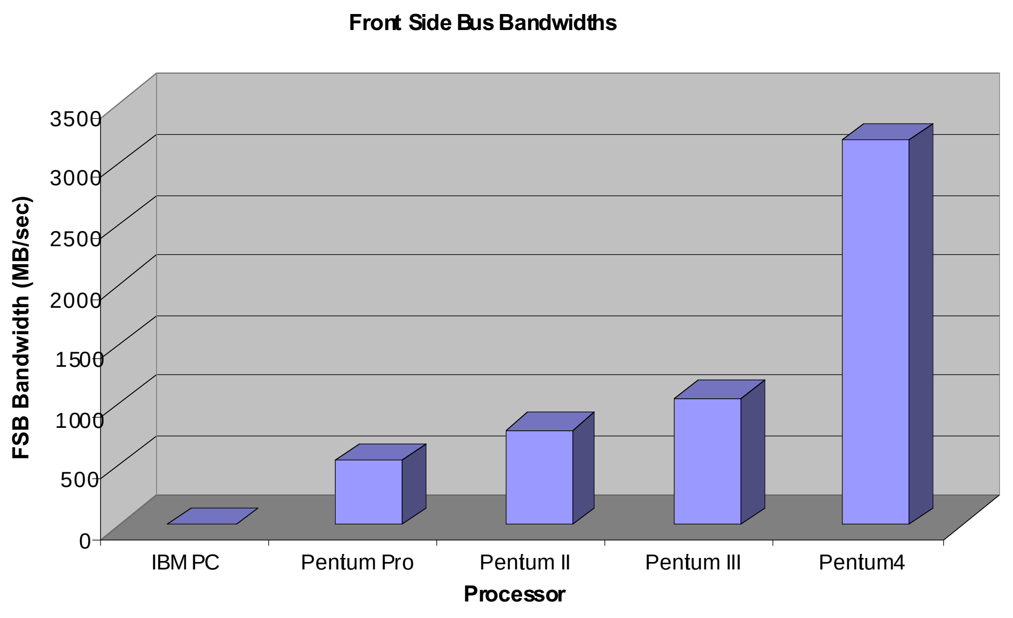


Figure 6: PC processor system bus bandwidths

PLATFORM IMPROVEMENTS DELIVER PERFORMANCE

As we have seen in the previous analysis of benchmarks, it is clear that system bus bandwidth is a limiter for the performance of the platform when high bandwidth is required by the application. This was foremost in the minds of the designers of the Intel Pentium 4 processor when developing the system bus. The system bus used in the Pentium 4 processor delivers unprecedented bandwidth for the PC platform, as can be seen in Figure 6. In addition, the system bus protocol has been improved to allow more deeply pipelined operations, memory prefetching, and glueless multiprocessing.

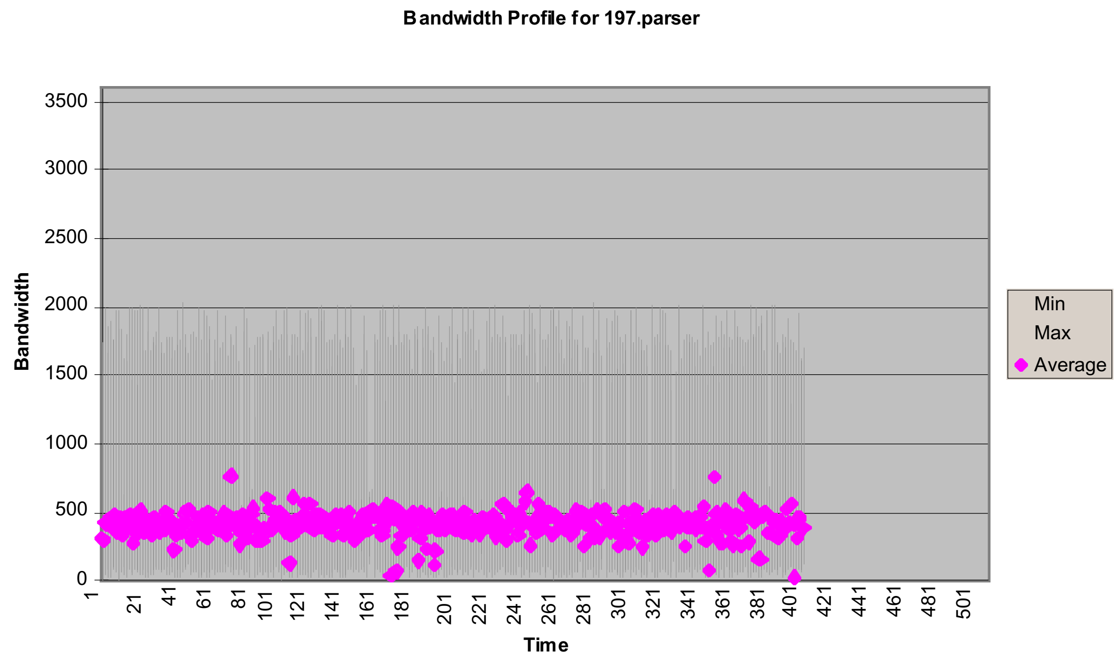


Figure 7: Pentium® 4 processor 197. parser bandwidth profile

The elevated execution speed and processor bus transfer rate of the Pentium 4 processor offer the opportunity for greatly enhanced application performance. This is illustrated by its performance on some benchmarks. To reiterate, one class of benchmarks would see reduced execution time solely because of processor frequency increases. On the other hand, different benchmarks might have found their achievable performance limited by the bandwidth on the processor bus without the advantage of the enhanced bus speed of the Pentium 4 processor. The first class of applications is typified by the 197.parser component of SpecINT, whose average and extreme bandwidths on a 1.4GHz Pentium 4 processor are plotted in Figure 7.

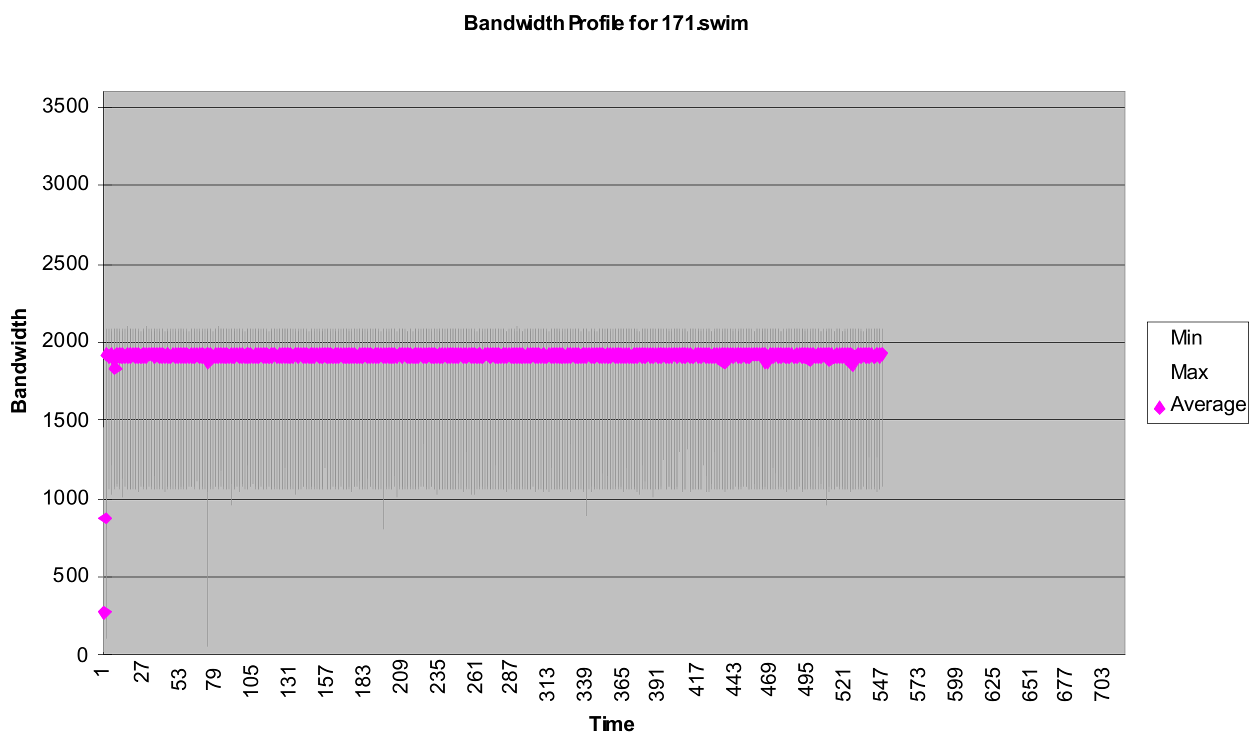


Figure 8: Pentium® 4 processor 171.swim profile

Note that the bandwidth required by the 197.parser increases dramatically from the Pentium III to the Pentium 4 processor. Where the average bandwidth of the Pentium III processor hovered around 300MB/sec, the bandwidth of the Pentium 4 processor requires around 400MB/sec during its reduced execution time.

Applications that were limited by the Pentium III processor bus experience the greatest speedup. 171.swim was shown in Figure 5 to be demanding memory bandwidth nearly equal to that of the Pentium III processor bus bandwidth. The corresponding results on a Pentium 4 system are shown in Figure 8. The bandwidth demanded by the Pentium 4 processor is nearly doubled over that of the Pentium® III processor. Since the Pentium 4 processor platform is able to satisfy the higher demand, the execution time of 171.swim is greatly reduced.

Note that systems with Pentium III processors simply did not support this high-bandwidth rate. Their maximum theoretical data transfer rate peaked at 1066MB/sec on a 133MHz bus.

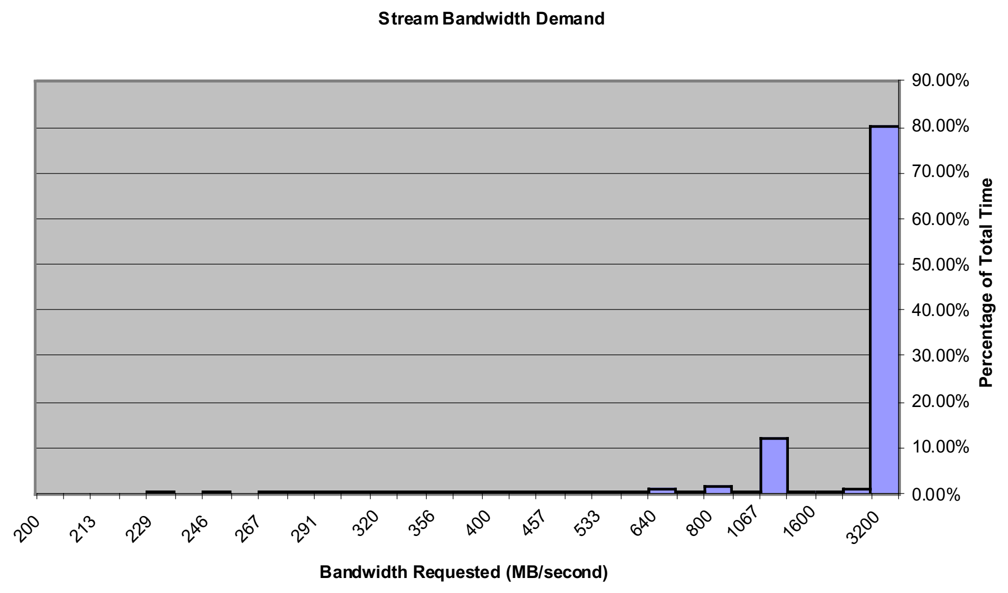


Figure 9: *STREAM* memory bandwidth demands

The analysis of the *STREAM* results is similar, but presented in a different format. Figure 9 shows that 80% of the memory requests are delivered at a 3.2GB/sec data rate. Recall too that this must be matched by the floating-point processing capability of the processor to sustain this level of performance. The high-performance floating - point unit of the Pentium 4 processor more than matches the computation task. As was the case with the *SPEC* trace results, any platform incapable of supporting the requested memory bandwidth of *STREAM* will have lower benchmark results. It should also be noted that *STREAM* analysis shows the parallelism of the Pentium 4 processor system bus, since the average number of outstanding memory requests over the duration of the benchmark is almost six, indicating that the 82850 memory controller is simultaneously processing that number of memory requests at all times. Figure 10 shows the *STREAM* results for the Pentium III processor at 1GHz and the Pentium 4 processor at 1.5GHz, demonstrating the advantages of both the high-performance execution engine of the Pentium 4 processor and the platform improvements of the system bus and memory controller.

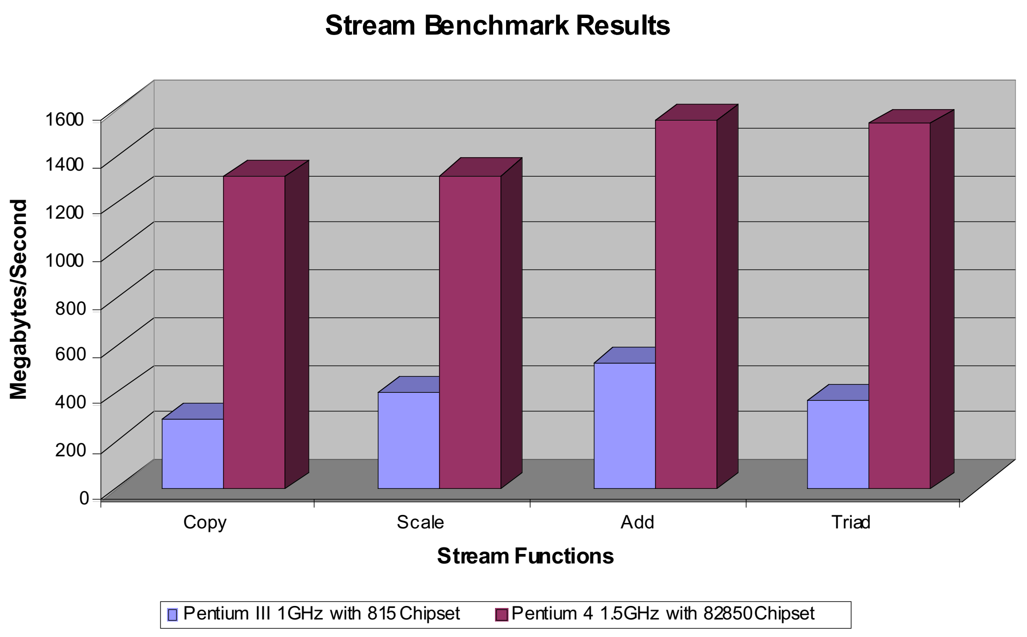
It should be noted that this methodology is independent of the memory technology. The two technologies presented here, SDRAM or PC133 and RDRAM merely represent two different technologies that are able to provide different memory bandwidths. This analysis could equally well be applied to different memory technologies, such as DDR SDRAM.

Should the memory demand of the application be unmet by the platform, the microarchitecture of the Pentium 4 processor can still compensate to some degree. Prefetching of memory requests allows the memory controller to more efficiently serve the requests. The high-performance pipeline of the Pentium 4 processor allows much higher processor frequencies to improve performance as the processor frequency increases.

CONCLUSION

It is not sufficient merely for the microprocessor to advance down its Moore's Law trajectory. In this paper, we have discussed the need for the platform components to move in concert with the microprocessors' technological advances. While there have been significant improvements in platform capabilities over time, the platform of the Intel Pentium 4 processor provides a significant leap forward for current platform capabilities. It is important to understand that it is the responsibility of the platform to meet the processor's demand for memory bandwidth with an appropriate supply. The system bus of the Pentium 4 processor provides a high-bandwidth channel for this brokerage function of the platform.

For the large body of existing applications, memory bandwidth demanded by the application is relatively low, so performance improvements can be realized with processor scaling alone. However, new applications and benchmarks require that the platform capabilities meet higher requirements to deliver performance. The Pentium 4 processor with its high-performance microarchitecture is capable of generating high demands for memory bandwidth, should the application require it. To fulfill



•

Figure 10: *STREAM* benchmark results

this demand, the platform components, primarily the system bus and memory connection, have been dramatically improved to supply this demand.

REFERENCES

1. McCalpin, John D., "Sustainable Memory Bandwidth in Current High-Performance Computers," October 12, 1995.
2. Hennessy, John L., and Patterson, David A., *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann Publishers, ISBN: 1558603298.
3. SPEC CPU2000 information can be found at http://www.spec.org/osg/cpu2000/.
4. SYSmark 2000 information can be found at http://www.bapco.com/sysmark2000primer.htm.
5. Information on the 82850 Memory Controller hub can be downloaded from http://developer.intel.com/design/chipsets/datashts/290691.htm.

AUTHORS' BIOGRAPHIES

R. Scott Tetrick is responsible for CPU Platform Architecture in the Desktop Products Group. He has been involved with bus development at Intel since 1979. He holds ten patents on platform architecture. While at Intel, he has developed platforms from single board computer systems to supercomputers and embedded controllers to multiprocessing servers. His e-mail address is stetrick@ichips.intel.com.

Blaise Fanning is a Platform Architect in DPG's chipset engineering group. He received his B.S. and M.S. degrees in computer engineering from Boston University in 1987. He joined Intel in 1997 and was the architect of the Intel840 workstation chipset. He is currently responsible for platform performance issues and developing next-generation I/O interconnects. His e-mail address is blaise.fanning@intel.com.

Robert Greiner is a platform architect with the DPG Oregon Architecture team. He helped develop the quad pumped system bus for the Pentium® 4 processor. He also contributed to the performance simulators for the Pentium 4 processor. He has worked on high-speed, scalable interconnect protocols for Futurebus+, MIT, and others. He graduated with a B.S. degree in Mathematics from Michigan State University. His e-mail address is rgreiner@ichips.intel.com.

Tom Huff is an architect in the Intel® Architecture Group in Oregon. He was one of the architects in the core team that defined the Streaming SIMD Extensions for the IA-32 architecture. He also worked on multimedia performance analysis for the Pentium® 4 processor. He holds M.S. and Ph.D. degrees in Electrical Engineering from the University of Michigan. His e-mail address is thuff@ichips.intel.com.

Lance Hacking joined Intel's IA-32 Architecture Group in Oregon in 1994 after completing undergraduate studies

at Brigham Young University. His focus on multimedia performance began with the Pentium® Pro processor, includes the Streaming SIMD Extensions defininition, and continues today with the current Pentium® 4 processor. His e-mail address is Lance.Hacking@intel.com.

Dave L. Hill joined Intel's DPG team in 1993, and has been the bus cluster microarchitect on P4 Willamette and Foster projects. Dave has 20 years industry experience primarily in high-performance memory system microarchitecture, logic design, and system debug. His e-mail address is dlhill@ichips.intel.com.

Srinivas Chennupaty is a processor architect in the Desktop Products Group in Oregon. He was one of the architects in the core team that defined the Internet Streaming SIMD Extensions for the IA-32 architecture. He is currently working on multimedia performance analysis for the Pentium® 4 processor. He holds a M.S. degree in Computer Engineering from the University of Texas at Austin. His e-mail address is chennu@ichips.intel.com.

David A. Koufaty received B.S. and M.S. degrees from the Simon Bolivar University, Venezuela in 1988 and 1991, respectively. He then received a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1997. For the last three years he has worked for the DPG CPU Architecture organization. His main interests are in multiprocessor architecture and software, performance, and compilation. His e-mail address is dkoufaty@ichips.intel.com.

Subba Palacharla joined Intel in 1998. Since then he has been a member of the DPG Architecture team working on performance evaluation of Willamette and Foster systems. Subba graduated from the Indian Institute of Technology, Kharagpur, India in 1991 with a B.Tech degree in Computer Science and Engineering. Subba received his Ph.D. degree in Computer Science from the University of Wisconsin at Madison in 1998. His e-mail address is subbarao@ichips.intel.com.

Jeff Rabe is a chipset architect in the Desktop Product Group, and was the lead architect for the 82850 chipset. He joined Intel in 1980, and has worked as a Yield Analysis engineer, Product Engineer, and Application Engineer prior to his chipset architecture role. His e-mail address is jeff.l.rabe@intel.com.

Mike Derr is an architect working on the I/O Controller Hub (ICH) product line within the Desktop Platforms Group. He has eight years of experience in Intel® chipset development and holds four U.S. patents related to chipset technology. He received a B.S.E.E. degree from Tennessee Technological University and an M.S.E.E. degree from the Georgia Institute of Technology. His e-mail address is mike.n.derr@intel.com.