A Performance Analysis of Pentium Processor Systems

Michael Bekerman
Technion, Israel Institute of Technology

Avi Mendelson
Technion, Israel Institute of Technology

Using a CPI metric, we analyze the performance of Pentium-based systems and examine their use of the processor's architectural features under different software environments. We break down the CPI into its basic constituents and examine the effects of various operating systems and applications on the CPI. This analysis indicates where the application spends its time during execution, giving designers a better understanding of design trade-offs and potential causes of performance bottlenecks.

The Pentium processor, a member of the Intel x86 family, offers several architectural improvements over Intel's i486 processor. In this article, we describe the architecture of the Pentium processor and analyze the performance of Pentium-based systems. Our objective is not to compare the Pentium's performance to that of other processors such as the i486 or other RISC (reduced instruction set computer) processors. Instead, we examine the use of the Pentium's architectural features by different software environments, with the goal of contributing to a better understanding of the design trade-offs and potential causes of performance bottlenecks of Pentium-based systems.

We use the CPI (cycles per instruction) criterion to analyze system performance. CPI measurement is an effective technique for comparing the performance of different applications on processors with different subsystems and clock frequencies but similar instruction set architecture. Added cycles per instruction is also a convenient way to express system speedup or slowdown. Breaking down the CPI into its basic constituents, we examine how different operating systems (Windows 3.1, Windows NT Version 3.1, Unix) and different classes of applications influence the CPI breakdowns of Pentium-based systems. CPI analysis shows where applications spend large periods of their execution time—information essential for the design of future architectural components and the correct division of resources between system or processor components.

One drawback of CPI analysis is that different compilers produce different codes, and, therefore, CPIs may differ slightly. In addition, we cannot directly compare the CPIs, or the CPI breakdowns, of various instruction set architectures (ISA) because different instruction sets result in different CPI behavior. (See the Related works box.)

Pentium architecture

Although the Pentium processor's instruction set is fully compatible with the i386/i486 processors, it has several new architectural features: superscalar dual-integer execution; an advanced branch target buffer (BTB) branch prediction mechanism; split cache (two 8-Kbyte on-chip caches for data and instructions); a pipelined, fast floating-point unit; a 64-bit external data bus interface; and an integrated performance-monitoring module. Figure 1 shows a general overview of the Pentium architecture, which we briefly describe here. A number of sources give more detailed descriptions.[1–7]

Execution pipes. The Pentium processor allows execution of two integer instructions in parallel through two five-stage pipelines (Figure 2). The following are the pipeline stages:

- *Prefetch (PF):* The processor prefetches instructions from the on-chip instruction cache. Since the processor has separate caches for code and data, prefetches do not conflict with data references for cache access.
- *Instruction decode (D1):* The processor attempts to decode two sequential instructions. It determines whether one or two instructions can be issued (sent to execution) according to the instruction-pairing rules (discussed later).
- *Address generation (D2):* This stage calculates memory reference addresses.
- *Execute (E):* This stage executes ALU operations and data cache accesses; thus, instructions requiring both an ALU operation and a data cache access will require more than one clock in this stage.
- *Write back (WB):* This stage enables instructions to modify the processor's state and complete execution. All exceptions and branch outcomes must be resolved before the instruction can advance to this final stage.

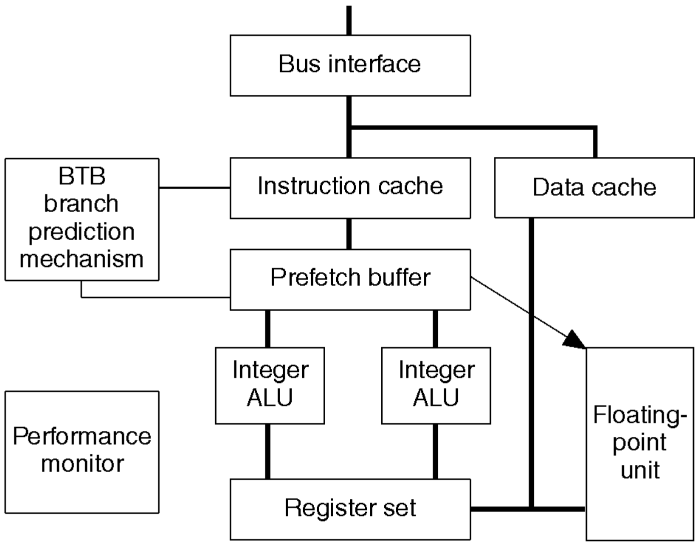

Figure 1 Pentium architecture.

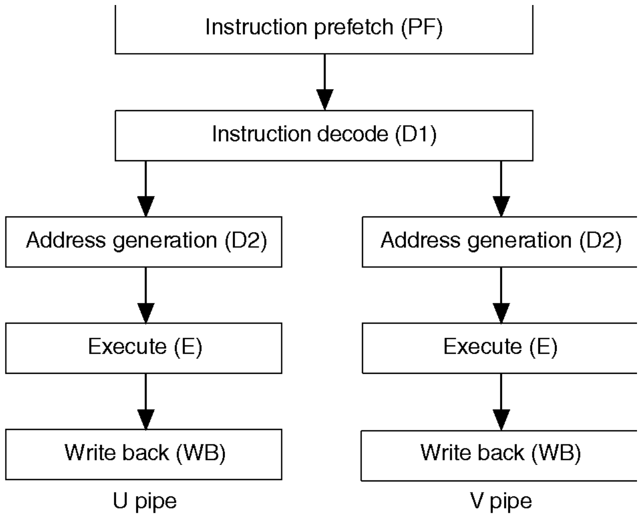

Figure 2 Execution pipes.

Related works

Several authors have published performance measurements of Pentium systems. Most of these studies give raw measurements such as the SPEC or SYSmark numbers for the systems. (SPECint92 and SPECfp92 are performance benchmarks[8] widely used by the Unix community to compare systems by running a set of integer and floating-point applications and measuring their execution times. SYSmark is a performance benchmark including typical PC applications.) However, only a few papers analyze use of the Pentium's new architectural features. Novitsky, Azimi, and Ghaznavi[9] study memory and bus subsystem design trade-offs and their impact on overall performance. Seymour[10] and Gwennap[1] report that a Pentium-based system can double the performance of a similar i486-based system running at the same frequency for SPECint92 applications and can even achieve a 3.5-fold speedup with SPECfp92 applications. Unfortunately, the SPEC92 benchmark mainly targets workstations and Unix applications, not Windows applications. As we show later, different software environments affect system performance differently.

The two parallel pipelines are called the U pipe and the V pipe. The processor always issues the first instruction of the pair to the U pipe; it issues the second instruction of the pair to the V pipe only if the instructions satisfy the following constraints, or pairing rules:

- Both instructions must be simple. Simple instructions include all common moves, ALU operations, increment/decrement, push/pop, near jumps, and calls. In addition, the U pipe can execute shift instructions. None of these instructions require microcode sequencing.
- There must be no register dependencies between the two instructions. In other words, a destination register of the U pipe instruction cannot be used as a source or destination of the V pipe instruction.
- Only instructions in the U pipe can use addressing modes that have both displacement and immediate values.
- Usually, only the U pipe can execute instructions with prefixes.

No memory dependencies are checked. If both pipes try to access the same cache bank, only the U pipe is allowed to access the data cache; the V pipe stalls and fetches the data in the next cycle. During their progression through the pipeline, instructions may stall due to certain conditions. No successive instructions can enter the E stage of either pipeline until instructions in both pipes have advanced to the WB stage.

Although floating-point operations usually utilize both pipes, one pair of floating-point instructions can execute simultaneously: any one-vector (that is, not requiring microcode) floating-point instruction with an FXCH instruction. This feature partially compensates for the x86's restrictive floating-point instruction organization, which references the floating-point registers only as a stack.

Branch target buffer. Today's architectures try to predict the direction of conditional branches and to speculatively fetch, decode, and execute instructions after the branch, while the actual branch direction is still unknown. This avoids stalling the prefetch process (note that the conditional branch is executed in the late E stage[4, 11, 12]). The Pentium processor uses the BTB to support the branch prediction mechanism. When a branch instruction is first taken (that is, when the branch changes the program control flow), an entry corresponding to this branch is allocated in the BTB. It associates the branch instruction address with the branch target address and the branch history information.

The decoder accesses the BTB in the D1 stage with the linear address of the instruction. If the BTB contains an entry corresponding to this address (a BTB hit), the BTB predicts a branch according to the branch history; otherwise, the processor fetches sequential instructions. If the BTB predicts the branch to be taken, the processor fetches the next instruction from the branch target address saved in the BTB. If the branch is predicted to be not taken, the processor continues to fetch sequential instructions. Later, when the processor calculates the branch condition and actually executes the branch instruction (that is, the branch is resolved), the prediction is validated. If the prediction is correct, the instructions may move to the WB stage (to change the processor's state). Otherwise, the processor flushes all instructions from the predicted path and fetches new instructions from the correct path.

Branches can be executed in both the U and the V pipes, and they distribute equally between the pipes.[6] There is no penalty for correctly predicted branches. On the other hand, the penalty for mispredicted branches depends on the execution pipe. In the U pipe the penalty is 3 cycles, since it is always known in the E stage whether the branch is taken or not. For mispredicted branches in the V pipe, the penalty is 4 cycles, since they can execute in parallel with the U pipe instruction. This instruction may modify the conditions' flags (this instruction precedes the branch in the program order). Therefore, the branch's outcome is known only in the WB stage.

The BTB contains 256 entries, organized as four-way set-associative. Each entry contains a branch instruction address, the branch target address, and 2 history bits. (The prediction algorithm used by the BTB is described by Lee and Smith.[11]) The two history bits represent four possible states: ST (strongly taken), WT (weakly taken), WNT (weakly not taken), and SNT (strongly not taken). When the history bits correspond to the first two states, the prediction is that the current branch will be taken; in the two remaining states, the prediction is that the branch will be not taken.

Figure 3 diagrams the state transitions after the branch is actually resolved. The letter on the arc represents the current branch direction: T is for taken and N is for not taken. For example, if the history bits correspond to ST state and the current branch was not taken, the bits will be updated to WT state. If this branch is also not taken the next time, its state will change to WNT and the prediction will change. The two middle states prevent the situation in which a single occasionally different branch behavior immediately changes the prediction. If the branch misses the BTB, it is assumed to be not taken, and the processor continues the sequential fetching. The new entry is allocated in the BTB if the missed branch was taken. (Otherwise, it is highly probable that the branch will not be taken the next time, too, and will be handled correctly by default.) The initial state of the new BTB entry is always ST.

Memory subsystem. The Pentium's memory subsystem aims to speed up access to data in memory with low additional cost, to support virtual memory, and to guarantee cache coherency. The processor uses separate instruction and data caches, which can be accessed simultaneously. Each cache is 8 Kbytes wide, has a two-way set-associative organization, and has a line size of 32 bytes. Both caches use physical tags to allow fast snooping (address detection) on the bus and LRU (least recently used) replacement policy. The instruction cache provides special support of split fetches: the ability to fetch a contiguous block of instructions even if it is split across two instruction cache lines.[4,6] The instruction cache tags have three ports: one port for snooping operations, and the other two for the split-fetch capability.

The data cache is organized into eight banks: the first 4 bytes of each cache line (bytes 0 through 3) are fetched from the first bank, the next four bytes are fetched from the second bank, and so on. Both pipes can access the data cache simultaneously, provided that the references are to different banks. Triple-ported data cache tags allow concurrent snooping and dual access by the two pipelines.[4,6]

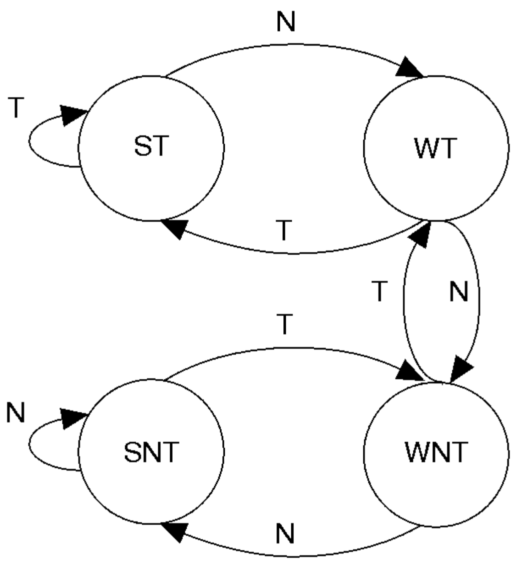

Figure 3 BTB prediction state transitions.

The Pentium processor is designed to work in a multiprocessor environment. Consequently, it uses the MESI10 cache coherency protocol to maintain coherency between the cache and the system I/O activities and between different caches in a multiprocessor system.

A translation lookaside buffer mechanism supports virtual memory. The TLB translates in hardware the virtual addresses used by programs into physical addresses required by caches and main memory. The processor uses the 64-entry data TLB, with a four-way set-associative organization, implemented as a dual-ported memory, to allow two simultaneous address translations by both pipes. The TLB translates 4K pages, which is standard for the x86 architecture.

There is an additional eight-entry, four-way set-associative, dual-ported data TLB for 4M pages.[6] It allows mapping of large address spaces used by graphics frame buffers and the operating system with a few 4M translations, preventing thrashing of a regular TLB. Both data TLBs use a pseudo-LRU replacement policy. The instruction TLB contains 32 entries, organized as four-way set-associative, eight sets per way. It implements a pseudo-LRU replacement policy.

Measurement methods

The following explains our statistics-gathering technique, method of breaking down the measured CPI into basic constituents, and empirical penalty metrics.

Statistics gathering. The Pentium processor has a pair of special-purpose counters for performance monitoring. Each counter can be programmed to count any one of 38 different available events. The counters work on the fly, in parallel with other processor logic, and do not interrupt or influence normal processor or system functioning and performance. The available-events list includes aspects of cache performance such as the number of cache accesses or misses, the number of BTB and TLB accesses and misses, pipeline stalls and flushes, dual instruction issues, and so on. (Ryan[13] provides the complete events list and explains how to program these counters.)

Since only two events can be measured simultaneously, we ran each benchmark several times to monitor different statistics. Results can vary by a few percent because operating system activities are counted as well. To minimize these changes, the benchmark we executed was the only user application in the system; furthermore, we always measured related events simultaneously.

The only parameter we could not measure directly with the monitoring counters was the amount of time the system stalled due to disk activities. For this purpose we used the Microsoft Windows NT Performance Monitor.[14] This tool allows counting of total disk access time during application execution. Apparently, this method overestimates disk waiting time because some disk activities may execute in parallel with the processor for example, page prefetching or operating system activities. In most cases, however, disk activities measured by the performance monitor matched the overall application's I/O activities; therefore, all disk activity is considered a penalty. More accurate measurements may result in a few percent decrease in the disk stall component of the execution time breakdowns.

These two performance-monitoring techniques have no significant effect on the system being measured: The hardware monitor operates in parallel with other processor activities, and the software tool samples the system at a frequency that enables it to be accurate enough with almost no interference in general system operation.

Breakdown method. With the Pentium performance-monitoring capability, we accurately measured the number of application instructions executed, processor cycles taken, and cycles wasted in pipe refilling after branch misprediction, waiting for cache refilling, TLB miss processing, and so on. Therefore, we can calculate the application's CPI and distinguish its contributors.

The term *penalty of any event* X indicates the number of cycles the processor stalls as the result of a single occurrence of an event *X*. The average per instruction contribution of an event *X* to the application's execution time is defined as

(Penalty of *X*) × (Number of occurrences of *X* during application execution)/(Number of application instructions)

The three dominant causes of processor stalling were memory access time, branch misprediction, and address generation interlock. Other causes had minor overall effect, and we grouped them as miscellaneous. The effect on the CPI of the second pipe and the imperfect dual-issue capability is complicated somewhat by pairing constraints. Performance counters can measure the real dual-instruction issues, but they cannot count instructions not executed in parallel due to constraints. Recall that only simple instructions can execute in parallel.

Furthermore, the two pipelines are not symmetric, so in many cases the possibility of instruction pairing can be determined only at runtime. Consequently, we can calculate neither the ideal pairing on the given code nor the penalty of pairing constraints. What we can measure is the second pipe's contribution to overall performance that is, the application speedup due to the V pipe. (The V pipe hardware also improves the execution time of complex microcoded instructions,6 but we don't count this feature as a V pipe contribution in this article.)

The measured CPI consists of the following components:

- *Basic CPI:* This is the CPI of a real Pentium processor machine on a given code without any additional cycle wastes—that is, infinite cache and 100-percent branch prediction. The basic CPI takes into account the real code pairing, not the ideal one. In other words, it assumes parallel execution of the known part of the instructions.
- *V pipe use:* We measured the extent of the V pipe's contribution to system performance. The superscalar capability is reduced by hardware limitations, complex microcoded instructions of the old code, and insufficient compilation techniques. Because the performance speedup, not the stall, was measured, the V pipe contribution is negative, reducing the CPI, in contrast to other contributors. In the absence of the second pipe, the application's basic CPI would increase by the value of this component.
- *Memory access penalties:* An average (per-instruction) system stall resulted from on-chip cache misses and TLB misses. This penalty includes the stall time incurred while waiting for all levels of memory hierarchy and TLB miss processing.
- *BTB misprediction:* As a result of wrong branch prediction, the processor flushes the instruction pipes, causing a penalty of 3 or 4 clocks for pipe refilling. We are interested in effective BTB misprediction that is, not-taken branches that have not been found in the BTB but have not caused pipeline flushes are counted as effective correct prediction. Taken branches that have not been found in the BTB always cause pipeline flushes and are counted as effective misprediction.
- *AGI (address generation interlock):* AGI occurs when the next instruction needs data (a register value or a memory location) for its address calculation, but this value has not yet been calculated by a previous or parallel instruction.
- *Miscellaneous:* I/O, segmentation overhead, cache bank conflicts, and so on.

Memory access penalties significantly affect system performance. Hence, we further partition the memory subsystem stalls into four components:

- *Second-level cache access penalty:* processor stall time while waiting for second-level cache response, caused by a first-level cache miss.
- *Main-memory access penalty:* caused by first- and second-level cache misses.
- *Disk access penalty:* an overhead of disk paging activities.
- *TLB miss processing:* the direct penalty for code and data TLB misses.

In contrast to the results presented here, some studies measure the performance of systems by looking only at a narrow set of parameters. As we show later, one must understand the entire picture to evaluate system performance. The following test cases support our claim:

- *Disk replacement:* In one of our tests, we moved from a fast SCSI (Small Computer Systems Interface) disk to a slow IDE (Integrated Device Electronics) disk. To our surprise, almost all the performance parameters, such as basic CPI, branch prediction, cache hit, and V pipe use, improved. However, the execution time was very poor and the instruction count was much higher. A close look at the disk driver code revealed that the IDE disk driver uses a busy-wait protocol, whereas the SCSI disk uses a special DMA (direct memory access). The IDE driver spent most of the time in a simple loop waiting for the information to arrive. During this small loop, most of the memory accesses resulted in cache hits, BTB prediction was accurate most of the time, average CPI decreased, but all the instructions contributed no useful work to the application.
- *PCI bus:* We found that using a fast PCI (peripheral component interconnection) bus does not always give the expected results. There were two major reasons for this: improper bus drivers,[3] and the PCI chip set was not fast enough to invalidate the second-level cache entries as a result of PCI bus transfer. Therefore, the second-level cache was forced to work as a write-through cache. Such caches do not perform as well as write-back caches, so overall performance may even be reduced.

Penalty metrics. We examined different computers, including X-desktop series PCs and PCI bus systems from various manufacturers. All the results and metrics presented here are for the system with the best execution time. As mentioned earlier, we had to execute each application several times to monitor different events. All runs were performed on the same system. This system consists of a 66-MHz Pentium processor, a 512-Kbyte second-level cache, 16 Mbytes of main memory, a PCI bus, and a PCI SCSI-II disk controller.

In the following discussion, we use several empirical penalty metrics. We used intricate tests to measure and calculate these values. (To investigate the penalties of second-level cache and main memory accesses, as well as TLB miss processing, we built a number of loops that accessed memory locations in patterns that isolated the required parameter. The V pipe contribution study required inspection of Pentium-optimized machine codes in conjunction with pairing rules.) The following list summarizes the penalties:

- A second-level cache access resulting in a cache hit has a penalty of 3 cycles. About 75 percent of accesses to the second-level cache (because of misses to the first-level cache) were cache hits.
- The main memory access penalty is 13 clocks. Only in the case of misses to both the first- and second-level caches is the memory accessed.
- The BTB misprediction penalty is 3.5 cycles. As previously described, the penalty is either 3 cycles if the mispredicted branch is in the U pipe, or 4 cycles otherwise. The branches are distributed almost equally between the pipes. Therefore, we charge each mispredicted branch a penalty of 3.5.
- The TLB miss-processing penalty is either 28 or 41 processor cycles, depending on whether the currently replaced entry is clean or dirty. We discuss TLB behavior and its time penalty in a later section.
- The V pipe contribution to overall system performance is an important issue, so we provide a step-by-step explanation of how we calculated these numbers: Examination of the Pentium's pairing rules[4,6] shows that the V pipe can execute only 1- or 2-cycle instructions. Each 1-cycle instruction executed in the V pipe saves 1 cycle. Each 2-cycle instruction in the V pipe paired with a 1-cycle instruction in the U pipe saves 1 cycle as well, because no other instructions enter the E stage before both instructions currently in the E stage proceed to the WB stage. Only 2-cycle instructions in both pipes can save more than a single cycle, but this is a rare situation according to our instruction profiles. In addition, paired instructions eventually may execute sequentially due to memory bank conflicts. Therefore, the effective savings of each V pipe instruction is one processor cycle, independent from the CPI.

Performance breakdowns

During our study we identified two categories of applications, each demonstrating different performance behavior. The first includes Unix-style applications represented by the SPEC92 benchmark suite.[8] These applications are CPU-intensivethat is, performance is mainly influenced by the speed of the core (the floating-point and the integer ALU units). These applications require relatively little operating system support, resulting in similar performance profiles on Unix and Windows NT.

The second set of applications is the SYSmark93 performance suite, consisting of Windows 3.1 native applications. These applications use system resources heavily. In such applications, the CPU is not always the performance bottleneck; their performance depends on the entire computer system. Later we show that their interference with the operating system is an important performance factor.

SPEC92 applications. We used six applications from the SPEC92 suite: four integer applications—compress, eqntott, espresso, and gcc—and two floating-point applications—ora and su2cor. Three of these applications have large working sets: compress, gcc, and especially su2cor. The others are CPU-bounded. We compiled the applications with the Pentium optimizing compiler.

Figure 4a illustrates the general breakdown of the earlier six SPEC92 integer applications. As explained earlier, an application's overall CPI consists of the basic CPI and the stalls encountered during execution. The column layers represent the various contributions to the CPI, as indicated by the key in the figure. The "Other" layer represents the AGI, BTB, and miscellaneous contributions.

Figure 4b shows the memory subsystem stall (the component represented by the third-from-the-bottom layer in Figure 4a), divided into four constituents: second-level (L2) cache, main memory, disk, and TLB miss-processing contributions per single instruction. For example, out of the total memory stall of 1.3 cycles per instruction for the compress application, 0.17 cycle is second-level cache waiting, 0.33 cycle is main memory waiting, and so on.

Figure 4c presents in more detail the three remaining stall components of the general breakdown: AGI stalls, branch misprediction penalty (BTB), and miscellaneous stalls. Again, the total CPI breakdown in Figure 4a subsumes both Figures 4b and 4c.

As expected, the basic CPI is different for integer applications and floating-point applications. While the integer applications' basic CPI is around 1, the floating-point applications' basic CPI approaches 3. The integer applications' basic CPI indicates that these applications take full advantage of the Pentium processor architecture. As Table 1 shows, about 30 percent of instructions execute in the V pipe, reducing the basic CPI by 0.3 cycle per instruction. Without the additional pipe, the basic CPI would be 1.3. One would expect a basic CPI of 0.5 from a superscalar dual-pipeline machine running RISC-like code produced by a modern compiler. Because only 30 percent of instructions execute in the second pipe, the expected basic CPI increases to 0.7. (The other 70 percent execute sequentially in the U pipe.) The Pentium processor uses CISC (complex instruction set computing) instructions that usually require more than a single execution cycle. These instructions, which the compiler cannot avoid completely, contribute the additional 0.3 cycle for each instruction.

Table 1 provides additional information about the causes of the stalls presented in Figure 4. This information is required for the performance analysis. The first parameter listed in the table, "D cache read hit rate," is the percentage of total data cache reads that resulted in cache hits. To measure its influence on performance, we must consider read frequency, or "D cache reads per instr." "Eff. BTB corr. pred." counts all the correct predictions made by the BTB and the not-taken branches predicted by default. (Branches that miss the BTB are predicted as not taken by default.) The names of the other parameters in the table are self-explanatory.

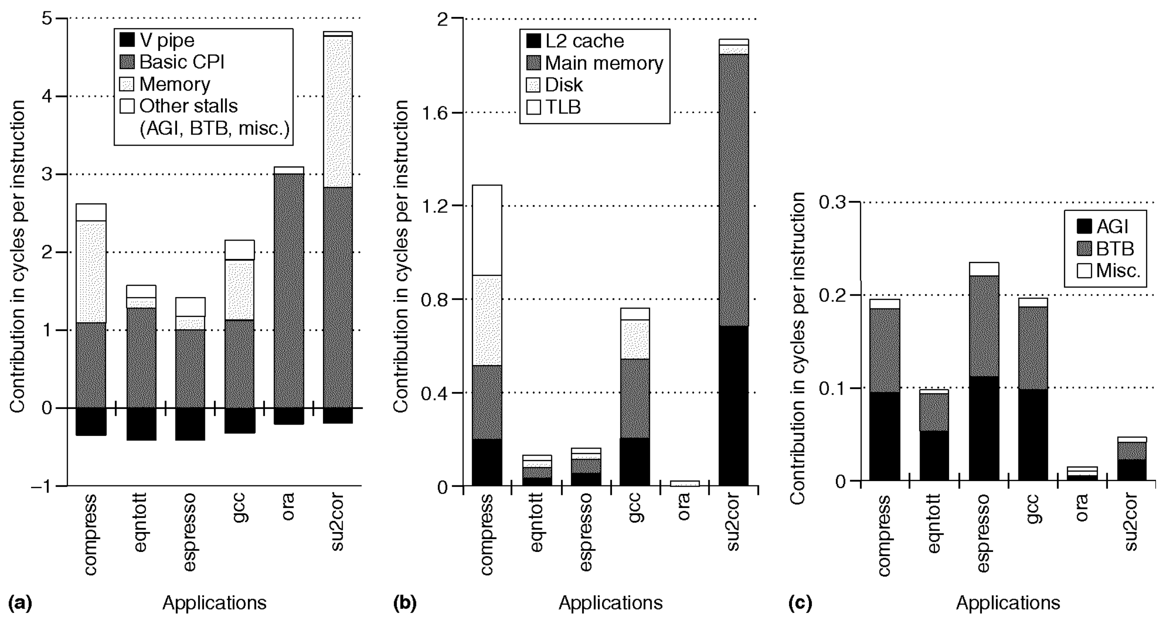

Figure 4 CPI breakdown of SPEC92 applications: overall CPI breakdown (a); memory subsystem stall breakdown (b); AGI, BTB, and miscellaneous stalls (c).

Table 1. Performance of SPEC92 applications.

Performance parameter	compress	eqntott	espresso	gcc	ora	su2cor
D cache read hit rate (%)	87.51	96.15	96.64	95.69	99.96	52.77
D cache reads per instr	0.36	0.22	0.30	0.33	0.47	0.50
D cache write hit rate (%)	80.05	93.96	92.17	78.32	99.91	51.97
D cache writes per instr	0.18	0.03	0.11	0.19	0.20	0.15
Data TLB hit rate (%)	97.95	99.84	99.87	99.85	99.99	99.91
I cache hit rate (%)	98.57	99.69	99.53	90.09	99.77	99.41
I cache reads per instr.	0.38	0.32	0.33	0.38	0.23	0.17
Instr. TLB hit rate (%)	99.93	99.99	99.99	99.81	99.99	99.95
Eff. BTB corr. pred. (%)	83.58	84.81	82.82	70.94	93.89	88.01
Instr. exec. in V pipe (%)	32.35	37.90	37.55	30.96	28.22	27.61
Overall CPI	2.62	1.57	1.41	2.15	3.03	4.75

Of the four integer applications in Figure 4a, espresso and eqntott have lower overall CPIs. These applications' V pipe use of near 37 percent (Table 1) is near optimal according to Intel's Pentium processor specification. The data cache read and write hit rates are very high for these applications—96 percent and 92 to 93 percent respectively (Table 1). The instruction cache hit rate is more than 99 percent. They also show relatively good BTB performance: the effective BTB correct prediction rate is in the range of 82 to 85 percent. This causes a pipe-refilling stall of about 0.1 cycle per instruction (Figure 4c). Thus, these applications' processor utilization is relatively good.

On the other hand, the gcc and compress applications have less favorable overall CPIs due to high memory subsystems stalls. Table 1 shows that the data cache read and write hit rates of these applications are low compared to eqntott and espresso. Both compress and gcc suffer from weak data locality; they use large data sets but access significant parts of them only once, resulting in a large number of "warm-up" misses. Hence, as Figure 4b shows, the memory subsystem stalls distribute almost equally between second-level cache waiting, main memory, disk-paging activities, and TLB miss processing. This adds 1.3 and 0.8 cycles per instruction to compress and gcc. The BTB performance of compress is similar to eqntott and espresso. The gcc code suffers from relatively low locality, manifested in a low effective BTB correct-prediction rate of 71 percent, which contributes almost 0.2 cycle per instruction to the total CPI (Figure 4c) and results in a low instruction cache hit rate.

We find different characteristics in the floating-point applications. The su2cor application has an overall CPI of about 4.7, and ora has an overall CPI of about 3 (Figure 4a). The basic CPI in both applications is very close to 3. The explanation of high basic CPI is that some of the floating-point instructions need more than one cycle in the execution stage to complete. However, only about 30 percent of the instructions are floating-point operations; the rest are integer instructions.

Although the floating-point unit is pipelined and can achieve a rate of one issued instruction per cycle for many floating-point instructions, very frequently a floating-point instruction is followed by a few integer instructions. In that case, the first integer instruction is effectively stalled for at least 3 cycles until the floating-point operation achieves the write-back stage. Therefore, pairing floating-point operations with at least some subset of integer operations will significantly boost performance by reducing the basic CPI. (Frequent integer operations in floating-point code are, for example, MOV, TEST, and cJMP.)

We explain the differences between the total CPIs of ora and su2cor by the differences in the use of their memory subsystems. While ora is strictly CPU-bounded, su2cor accesses large matrices, thereby causing a thrashing of the data cache. Table 1 lists unusually low data cache read and write hit rates of about 52 percent for su2cor. Figure 4b shows that the thrashing of the first- and second-level caches by the su2cor application expends as much as 1.1 cycles per instruction for main memory accesses.

The two floating-point, numeric applications demonstrate a different control flow locality than integer applications. Their instruction cache hit rate is very high, and, more notably, the BTB prediction rate reaches 88 percent in su2cor and 94 percent in ora, while the integer applications' BTB prediction rate is 82 to 84 percent. The good predictability of branches in these applications reduces the pipe-refilling stall to less than 0.02 cycle per instruction.

Windows-based applications. Most x86-family processors are used in IBM PC systems, many running Microsoft Windows-based applications. In this section we provide the CPI breakdown of the SYSmark93 performance suite, consisting of typical Windows environment applications. This benchmark includes popular Windows applications such as Excel 4, Lotus 1-2-3, and Word 2.

All the applications discussed here, except 32Excel, are Windows 3.1 native applications. They are regular versions of the applications, not recompiled with new compilers nor with Pentium optimizations. These applications run under a 16-bit subsystem of Windows NT. The 32-bit Excel (beta version) is the only application compiled specifically for Windows NT.

Figure 5 presents the performance breakdowns of three SYSmark93 applications: Excel in 16-bit and 32-bit versions, Lotus, and Word. The first four columns from the left represent these applications as executed on Windows NT Version 3.1. The two rightmost columns show the breakdowns of two of these applications (labeled with the extension 3.1) as run on Windows 3.1.

Table 2 summarizes additional performance parameters for the applications in Figure 5. The two rightmost columns list parameters of Excel and Lotus executed on Windows 3.1; all others are for Windows NT.

The overall CPI taken from direct measurements of the system is between 3.5 and 5, and the performance breakdown is very different from that of the Unix-style programs. These numbers appear to be surprisingly high, triggering us to further investigate the causes.

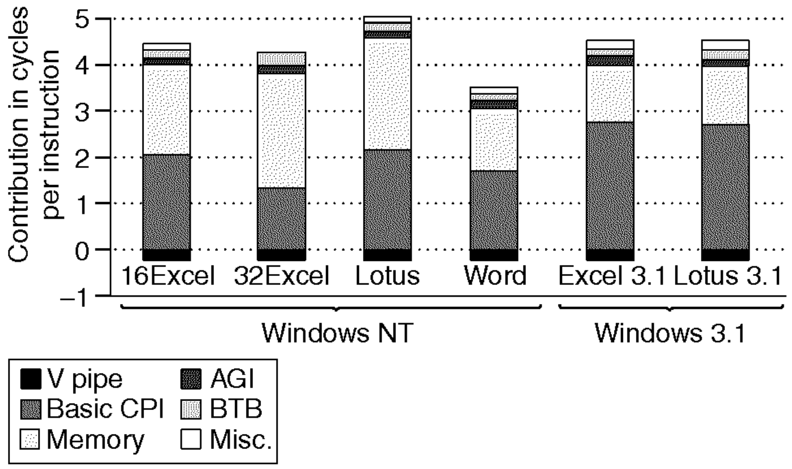

Figure 5 Overall CPI breakdowns of SYSmark93 applications.

Table 2. Performance of SYSmark93 applications.

Performance parameter	16Excel	32Excel	Lotus	Word	Excel3.1	Lotus3.1
D cache read hit rate (%)	91.48	91.18	91.24	93.79	93.73	94.02
D cache reads per instr.	0.42	0.42	0.42	0.45	0.45	0.49
D cache write hit rate (%)	84.04	83.61	83.99	83.74	87.71	88.99
D cache writes per instr.	0.28	0.27	0.28	0.24	0.28	0.31
Data TLB hit rate (%)	98.43	97.93	97.54	98.80	99.89	99.70
I cache hit rate (%)	73.53	74.07	71.24	81.60	75.46	79.46
I cache reads per instr.	0.33	0.36	0.34	0.36	0.40	0.46
Instr. TLB hit rate (%)	97.79	97.71	96.99	98.59	99.62	99.70
Eff. BTB corr. pred. (%)	63.51	60.40	61.51	63.55	77.52	81.72
Instr. exec. in V pipe (%)	18.61	20.20	17.51	17.98	16.62	16.22
Overall CPI	4.46	4.25	5.03	3.49	4.63	4.61

The basic CPI of most SYSmark93 applications is around 2; the overall CPI is more than double the basic CPI; in other words, the processor is stalled more than half the time. One reason that the basic CPI of the SYSmark93 applications is twice that of the basic CPI of the SPEC integer applications is the use of the old compiler. Such compilers use more CISC-type, complex, microcoded instructions, such as MOVSb (move string byte), taking dozens or even hundreds of cycles for a single instruction. (A similar RISC architecture will break such instructions into a loop of MOV instructions that will improve the CPI but does not automatically speed up the system.)

An additional cause of the SYSmark code's high basic CPI is ALU-memory instructions. Arithmetic instructions with an operand in memory take at least 2 cycles in the execution stage, and with both source and destination in memory, even 3 cycles. Due to the very small register set (eight registers) of the x86 architecture, such instructions are very frequent.

The V pipe parameter indicates system speedup due to superscalar capability—that is, dual-instruction execution. Only 16.22 to 20.20 percent of instructions execute in the V pipe (Table 2). The main reason for such low V pipe use is frequent complex, microcoded and memory-to-memory instructions that prevent a dual issue. To improve the performance of these applications, the Pentium needs less-constrained pairing rules.

There are other reasons for the system's inability to make significant performance gains from the Pentium's V pipe logic on Windows applications. The large overall CPI of these applications drastically reduces the significance of the V pipe contribution. The 17 percent V pipe contribution reduced the total execution time by about 0.17 cycle per instruction. Given that the total CPI is more than 4 cycles, the V pipe improves the total execution time by only 4 percent.

The two dominant components of the stall in both Windows NT and Windows 3.1 are the memory subsystem and BTB misprediction penalties. We are using a relatively high-end computer with a 512-Kbyte second-level cache, fast memory, and SCSI-II disk controller on the PCI bus (although the second-level cache works only in the write-through mode due to PCI bus interface problems). We expect that the relative penalty of the memory subsystem under a low-end computer will be even higher than that presented here because of weaker memory subsystem components.

The potential benefit of enhancing the architecture with a superpipeline or a large number of execution pipes depends on the quality of the branch prediction mechanism. We found that the branch prediction mechanism for the SYSmark93 application on Windows NT successfully predicted the direction in only 60 to 64 percent of the branches, adding to each instruction a pipe-refilling stall of almost 0.3 cycle. The 60 to 64 percent rate is very close to the static "always taken" policy performance. Thus, the prediction improvement of the dynamic, relatively complicated Pentium BTB is insufficient under this environment.

Memory subsystem. Because the memory subsystem is one of the potential bottlenecks in computer systems, we pay special attention to it. Figure 5 shows the SYSmark applications' total memory stall (third layer from the bottom). Figure 6 breaks this stall into contributions of different memory subsystem components. (As in Figure 5, the first four columns from left show applications running on Windows NT; the right two columns, Windows 3.1.) A refinement of stalls caused by the memory subsystem reveals that SYSmark applications lose about 1 cycle per instruction for accessing information in the second-level cache and main memory. Although Windows NT and Windows 3.1 use different memory management mechanisms, the contribution of the second-level cache and main memory to overall CPI is almost identical for all the applications and for both operating systems. The disk contribution differs from one application to another, but the same magnitude of disk stalls was found for the same applications running on both Windows 3.1 and Windows NT.

TLB miss processing under Windows NT was very expensive, contributing about 1 cycle per instruction (Figure 6). This observation contradicts other reports, which assumed that TLB miss-processing time is negligible. We found this assumption correct for most of the SPEC92 and Windows 3.1 applications (as shown in Tables 1 and 2); however, TLB miss processing introduces a large stall in the Windows applications running under Windows NT. The TLB hit rates in Table 2 point to TLB miss rates of 2 to 3 percent for these applications, adding as much as 1 cycle per instruction.

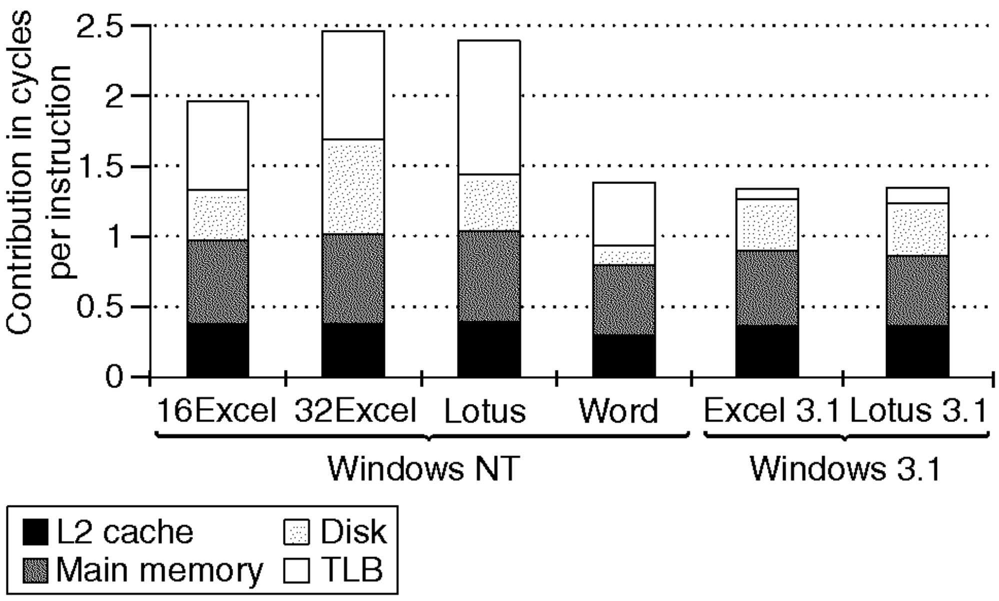

Figure 6 Breakdown of SYSmark93 applications' memory subsystem stall.

Windows NT uses a different software technology than Windows 3.1. Windows NT is based on a microkernel technology, which provides much of the operating system support by means of various subsystems, each implemented as a multithreaded server. (Threads are defined in Windows NT as lightweight processes executing in a partially shared address space.) This structure changes the program's control flow and reduces the locality of both data and instruction references. This characteristic does not greatly affect programs that do not require substantial operating system support, such as SPEC92 applications. Apparently, the new technology does not affect applications that run on Windows 3.1. On the other hand, the same applications running on Windows NT must loosely interact with the Windows NT services (subsystems) and thus suffer a significant TLB miss penalty.

Multiple threads of the same task execute in a shared address space, leading to the assumption that this should ease the penalty for process switches. However, each thread has its own working set, which must be replaced on a thread switch. Since the number of entries in the caches and the TLB is limited, frequent thread switches reduce the efficiency of the cache subsystem and the TLB.

Upon examining basic performance parameters of the TLB, we obtained some unexpected results. We found that the TLB miss penalty depends on whether the replaced entry was dirty or clean. To examine the TLB miss penalty, we developed a program that accesses an array large enough to cause a TLB miss on each access, but the references were in such a pattern that after the first pass, all of them were in the on-chip cache. We constructed the program from a single loop that accesses a single address on each memory page of the array.

We executed the loop enough times to reduce the effect of the cold start. We discovered that if we accessed all the array elements for read only (that is, all the TLB entries remained clean), the TLB miss penalty was 27.8 cycles—about two main memory accesses. On the other hand, when we accessed the same array for write operations (all the TLB entries were dirty), the TLB miss penalty was 41 cycles.

Comparison of Windows 3.1 and Windows NT. All the tested applications ran slower on Windows NT than on Windows 3.1 because Windows NT applications have 5 to 20 percent more instructions contributed by the operating system. These additional instructions, written according to today's programming concepts and compiled with new compilers, are better tuned for new architectures and have lower CPIs. This explains the slightly lower basic CPI for applications running on Windows NT than for the same applications on Windows 3.1.

The two environments' main difference is the much lower locality manifested by applications on Windows NT. Many Windows NT subsystems are written in C++. This object-oriented language preserves weak spatial and temporal locality, thereby reducing the cache hit ratios and lowering branch prediction accuracy. Inspection of Table 2 shows that data and instruction cache miss rates are higher on Windows NT. The drastic change, however, is in TLB (both data and instruction) hit rates and the BTB correct prediction rate.

Note the differences between the older, 16-bit version of Excel and the Windows NT native version. One possible explanation for the unexpectedly low speed of Pentium-based machines on Windows 3.1 applications is the execution of the 16-bit code. But our experiments indicate that 32Excel, a full 32-bit application, gives only a slight speedup. Figure 5 shows that although the basic CPI decreased from 2 in the 16Excel to 1.4 in the 32Excel, the memory subsystem stall increased; the overall CPI (Table 2) remained almost unchanged. We conclude that code characteristics determine processor performance and overall CPI, but segmentation overhead of the 16-bit code is only one of these characteristics.

Comparison of SYSmark93 and SPEC92 code characteristics. Our study examined the causes of the SYSmark93 applications' much lower performance than the SPEC92 applications. Our conclusions involve the following characteristics of Windows 3.1 native applications:

- High memory reference rate: around 1.15 memory references (data cache reads, data cache writes, instruction cache reads) per instruction compared to 0.8 in SPEC92 (Tables 1 and 2).
- Weak spatial and temporal locality: On average, SYSmark93 applications have much lower first- and second-level cache hit rates than SPEC92 applications; SYSmark93 applications' BTB performance is also lower.
- High percentage of noncacheable references (display references): This factor is most important in the absence of a graphics accelerator card.
- Greater use of prefixes than in SPEC92 applications.
- More microcoded instructions: the Windows applications were not compiled with today's compilers.
- The Windows applications were not compiled with a Pentium optimizing compiler; recompilation will increase pairing and decrease some stalls, such as AGI.

All these factors primarily influence basic CPI; correspondingly, the basic CPIs of SPEC92 applications are much lower than those of SYSmark93. The Pentium processor was greatly influenced by RISC concepts, and full use of its features is achieved only in executing RISC-style code. Good examples are the espresso and eqntott applications, which have basic CPIs of about 1 and almost no stalls. The V pipe use (about 38 percent) of these applications is better than that of other applications. Furthermore, pairing improves total CPI by more than 30 percent—a very high speedup if we consider the Pentium's complex pairing rules. Recall that in SYSmark93 applications the V pipe contribution to overall performance was as low as 4 percent due to low pairing and relatively high overall CPI.

Findings and implications

We distinguished between two basic classes of applications: Unix-style workstation applications, represented by SPEC92, and Windows-style PC applications, represented by SYSmark93. We found that SPEC92 applications behave similarly in Unix and Windows NT environments because of relatively low interference with the operating system. These applications can almost fully utilize the Pentium's features; the basic CPI is as low as one can expect for CISC architectures (1 for integer code and 3 for floating-point). The V pipe is well used and branch prediction is relatively accurate.

For most applications, the cache was sufficiently large, and, therefore, the contribution of the memory subsystem to the CPI was reasonable (in the system we used). Some applications, such as compress and su2cor, exhibited a higher degree of stalling due to the memory subsystem. We attribute this to the large working set used; no reasonable-size on-chip cache memory can hold all the data. Thus, at least with current technology, the replacement misses (resulting from still-to-be-used entry replacement because of insufficient cache size) cannot be avoided. However, a large second-level cache (more than one Mbyte) can improve performance, since it captures most of the working set of the considered applications.

The Windows-based applications display different characteristics; most of them cannot take full advantage of many of the Pentium features. The basic CPI of these integer-based applications is relatively high (around 2). Various stalls contribute the same order of magnitude to the overall CPI; that is, the processor is stalled for about half of its execution time. The main contributors to these stalls are the memory subsystem and the BTB misprediction overhead.

One of the main differences between the two application classes is their pattern of access to the memory hierarchy. For the SYSmark applications, the influence of memory bandwidth, response time, and bus speed are much more critical than for the SPEC92 applications. The SYSmark applications impose different system design requirements. One possible way to enhance the performance of such software environments is the use of better graphic accelerators. In this case the accelerator, rather than the CPU, will perform many of the complex instructions and noncacheable references.

Conclusion

We believe that Windows applications require a new approach to cache management. As we have seen, Windows applications exhibited relatively low locality, resulting in poor cache use. Future investigation of the memory access patterns of these applications should provide new memory management techniques. The effect of cache and TLB hit ratios on overall system performance will be more significant for future operating systems and applications, such as multimedia applications, which will use object-oriented languages and 64-bit address spaces.

We showed that for some applications the Pentium processor's branch prediction rate was low. This rameter is very important for the future design of Intel architectures. Improved branch prediction accuracy is vital for superscalar and superpipeline machines. Further research is required to develop a new branch prediction mechanism that offers similar accuracy for Windows applications as for Unix applications.

A significant portion of the Pentium processor die is spent on implementing superscalar capability. Unfortunately, several factors prevent the system from gaining much performance by using the V pipe: 1) The Pentium's pairing rules are highly constrained and complicate the compiler optimizations, 2) complex instructions cannot execute in parallel, and 3) when the basic CPI is large, as in the SYSmark applications, the contribution of parallel instruction execution is relatively low. There is a trade-off between maximal speedup of microcoded instructions and improved parallelism. Microcoded-instruction speedup requires the use of both pipes by the microcoded instruction, whereas to improve parallelism, the microcoded instruction should execute in only one pipe in parallel to some other instruction in the other pipe. Taking into account that today's compilers use fewer and fewer microcoded instructions, the parallel approach may be better because it does not interrupt the normal instruction advance through the pipes.

One possibility is to enhance only one pipe for fast execution of microcoded instructions, leaving the second for only simple instructions, and to let the compiler so order the instructions. Execution of complex and microcoded instructions in parallel will turn the save of each parallel instruction into more than one cycle. Therefore, its contribution will be much more appreciable than the contribution of pairing growth only. Another possibility is the pairing of floating-point instructions with integer instructions, since in most cases there are integer instructions (MOV, TEST, cJMP) between the floating-point instructions.

The new generations of operating systems and applications will be based on a multithreaded, client-server approach. Recently, we started looking at the effect of these features, particularly in the Windows 95 operating system, upon the overall performance of modern computer architectures. We are also looking for possible improvements in current wide-superscalar, out-of-order computer architectures to enhance the performance of Windows-based applications.

Acknowledgments

We thank Adi Golbert, Roni Ronen, and Nathaniel Hoffman from the Intel Architecture Group in Haifa, Israel, for reviewing this article and providing useful comments. We thank Intel Corporation for supporting this research and for generously supplying us the performance-monitoring software.

References

1. L. Gwennap, "Pentium Approaches RISC Performance," *Microprocessor Report*, MicroDesign Resources, Mountain View, Calif., Mar. 29, 1993, Vol. 7, No. 4, p. 1.
2. *Pentium Processor User's Manual, Volume 1: Pentium Processor Data Bo*ok, Intel Corp., Mt. Prospect, Ill., 1993.
3. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual,* Intel Corp., Mt. Prospect, Ill., 1993.
4. D. Anderson, and T. Shanley, *Pentium Processor System Architecture*, MindShare Press, Richardson, Tex., 1993.
5. D. Alpert, and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro,* Vol. 13, No. 3, June 1993, pp. 11–21.
6. B. Case, "Intel Reveals Pentium Implementation Details; Architectural Enhancements Remain Shrouded by NDA," *Microprocessor Report*, Mar. 29, 1993, Vol. 7, No. 4, p. 9.
7. B. Case, "Pentium Extends 486 Bus to 64 Bits; Higher Frequencies, New Features Improve Performance," *Microprocessor Report*, Apr.19, 1993, Vol. 7, No. 5, p. 10.
8. K.M. Dixit, "New CPU Benchmark Suites from SPEC," *Proc. Compcon, Spring 1992*, IEEE Computer Society Press, Los Alamitos, Calif., pp. 305–310.
9. J. Novitsky, M. Azimi, and R. Ghaznavi, "Optimizing Systems Performance Based on Pentium Processor," *Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 63–72.
10. J. Seymour, "Pentium: The Second Wave," *PC Magazine*, Jan. 1994, pp. 110–159.
11. J. Lee, and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Vol. 17, No. 1, Jan. 1984, pp. 6–22.
12. D.A. Patterson, and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990.
13. B. Ryan, "Inside the Pentium," *Byte*, May 1993, pp. 102–104.
14. R. Blake, *Optimizing Windows NT, Windows NT Resource Kit, Vol. 3,* Microsoft Press, Redmond, Wash., 1993.

Michael Bekerman is currently a research fellow at IBM Science and Technology, Haifa, Israel, where he participates in PCI-based systems design. His research interests include computer architecture, parallel and multithreaded processing, and performance evaluation.

Bekerman received the BSc and MSc degrees in electrical engineering from Technion, Israel Institute of Technology, working on multithreaded processor architecture development during his MSc studies.

Avi Mendelson is cohead of the Parallel Systems Laboratory and a lecturer in the Department of Electrical Engineering, at Technion, Israel Institute of Technology. His main research interests are computer architectures, operating systems, performance evaluation, and fault-tolerant systems.

Mendelson received the BS and MS degrees in computer science from Technion and the PhD from the Electrical and Computer Engineering Department, University of Massachusetts, Amherst. He is a member of the IEEE Computer Society.

Direct questions about this article to the authors at Dept. of Electrical Engineering, Technion, Israel Institute of Technology, Technion City, Haifa, Israel 32000; bekerman@psl.technion.ac.il or mendlson@ee.technion.ac.il.