Intel® Technology Journal

Intel® Pentium® 4 Processor on 90nm Technology

The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology

The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology

Darrell Boggs, Desktop Platforms Group, Intel Corporation
Aravindh Baktha, Desktop Platforms Group, Intel Corporation
Jason Hawkins, Desktop Platforms Group, Intel Corporation
Deborah T. Marr, Desktop Platforms Group, Intel Corporation
J. Alan Miller, Desktop Platforms Group, Intel Corporation
Patrice Roussel, Desktop Platforms Group, Intel Corporation
Ronak Singhal, Desktop Platforms Group, Intel Corporation
Bret Toll, Desktop Platforms Group, Intel Corporation
K.S. Venkatraman, Desktop Platforms Group, Intel Corporation

Index words: Pentium® 4 processor, Hyper-Threading Technology, microarchitecture

ABSTRACT

This paper describes the first Intel® Pentium® 4 processor manufactured on the 90nm process. We briefly review the NetBurst® microarchitecture and discuss how this new implementation retains its key characteristics, such as the execution trace cache and a 2x frequency execution core designed for high throughput.

This Pentium 4 processor improves upon the performance of prior implementations of the NetBurst microarchitecture through larger caches, larger internal buffers, improved algorithms, and new features. This processor also implements Hyper-Threading Technology, which is the ability to simultaneously run multiple threads, allowing one physical processor to appear as two independent logical processors. This technology is another means of providing higher performance to the end user. We discuss how this processor not only maintains support for this key technology but also increases the benefit seen due to Hyper-Threading Technology.

We also describe 13 new SSE3 instructions that have been added to the IA-32 instruction set and are implemented for the first time on this processor. These instructions can be used in multimedia algorithms, such as motion estimation, and for complex arithmetic. Additionally, two new instructions are added for improving thread synchronization. To conclude, performance data are presented that show the benefit of this Pentium 4 processor over prior implementations on key applications and benchmarks.

INTRODUCTION

The first Intel Pentium 4 processor manufactured on the 90nm manufacturing process contains 125 million transistors with a die size of 112mm². It builds upon the NetBurst microarchitecture that forms the foundation of prior Pentium 4 processors. Like its predecessors, this processor is designed to provide the end user with new levels of performance, enabling compute-intensive tasks to be undertaken by conventional desktop processors. One means of achieving this performance is by designing the processor to run at a high frequency. The frequency of a processor is a key component to determining overall performance, as the frequency determines the rate at which the processor can process data. We have extended the original Pentium 4 processor pipeline to enable this processor to reach

higher frequencies than is possible with the original pipeline. Additionally, as the frequency of the processor continues to increase, the amount of time spent waiting for data to be retrieved if they are not located in the processor's caches is becoming a larger and larger percentage of overall execution time. This effect reduces the performance impact of continually increasing the

processor frequency. To alleviate this problem, several features are implemented to increase the number of times that data will be present in the caches. With these and other features, including a set of new instructions, the Pentium 4 processor is able to achieve new heights in performance.
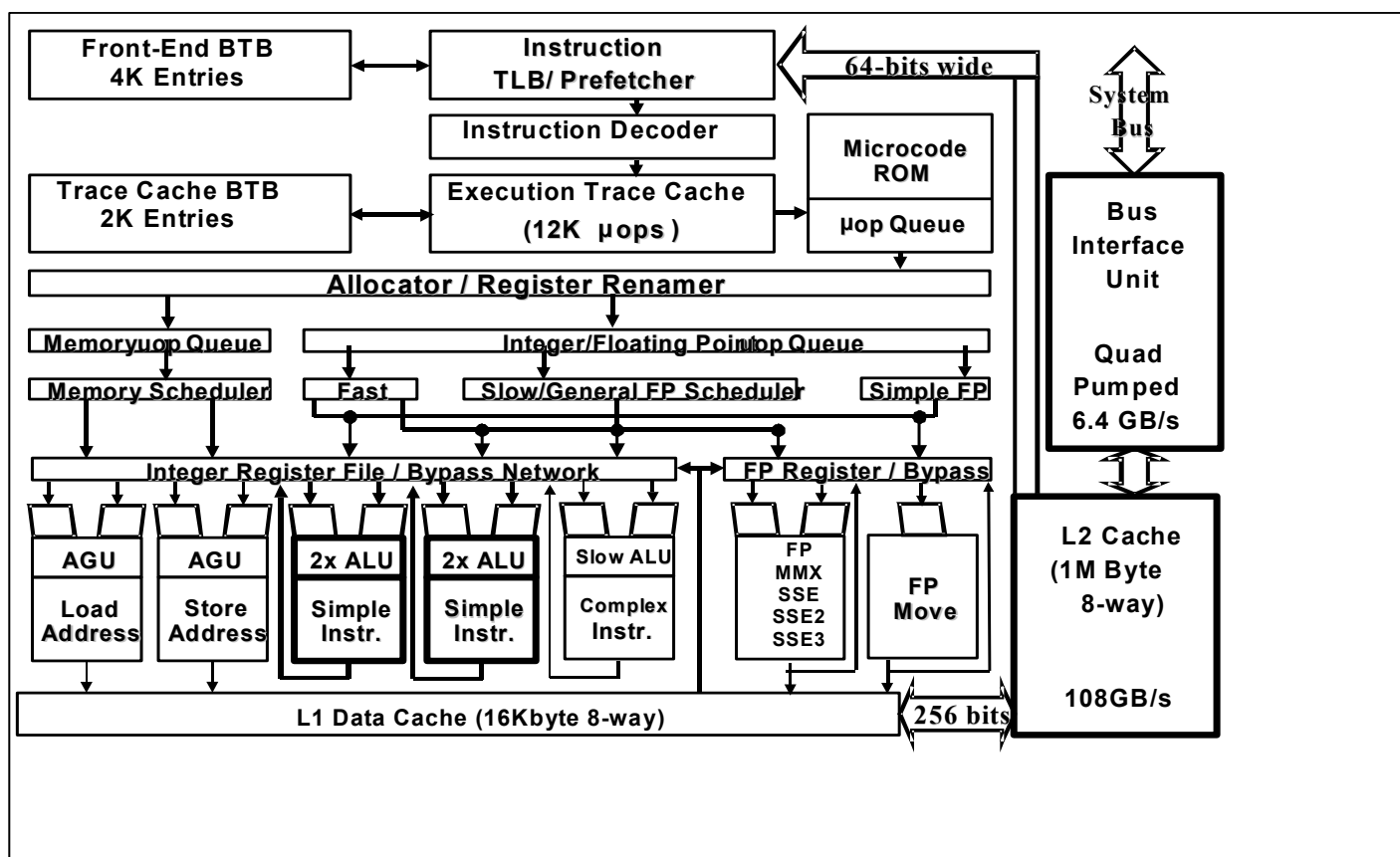

Figure 1: Block diagram of the Intel® Pentium® 4 processor

NETBURST® MICROARCHITECTURE OVERVIEW

The NetBurst microarchitecture is the basis for the latest version of the Intel Pentium 4 processor. Elements of this microarchitecture include an Execution Trace Cache, an out-of-order core, and a Rapid Execution Engine [1]. This implementation also contains store-to-load forwarding enhancements that were introduced in previous implementations. Figure 1 depicts the block diagram for the Pentium 4 processor.

Execution Trace Cache

The NetBurst microarchitecture has an advanced instruction cache called an Execution Trace Cache. This cache stores decoded instructions in the form of uops rather than in the form of raw bytes such as are stored in more conventional instruction caches. Once stored in the trace cache, uops can be accessed repeatedly just like a conventional instruction cache. Storing uops instead of bytes allows the complicated instruction decoding logic to be removed from the main execution loop.

In addition to removing the cumbersome decode logic from the main execution loop, the Execution Trace

Cache takes the already decoded uops from the instruction decoder and assembles or builds them into program-ordered sequences of uops, called traces. It packs the uops into groups of up to six uops per trace cache line and these lines are combined to form traces. These traces consist of uops from the sequentially predicted path of the 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 instruction are thousands of bytes apart in the program. Thus, both the branch and its target instructions can be delivered to the out-of-order core at the same time. Conventional instruction caches typically provide instructions up to and including a taken branch in a given clock cycle but no instructions following the branch. 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 of these instruction delivery delays.

The trace cache is able to deliver up to three uops per clock cycle to the out-of-order core. Most instructions in a program are fetched and executed from the trace cache. Only when there is a trace cache miss does the machine fetch and decode instructions from the unified second-level (L2) cache. The Execution Trace Cache on the Pentium 4 processor can hold up to 12K uops and has a hit rate similar to an 8 to 16 kilobyte conventional instruction cache.

Out-of-Order Core

The Execution Trace Cache provides the out-of-order core with a stream of uops to prepare for the Rapid Execution Engine to consume. The main responsibility of the out-of-order core is to extract parallelism from the code stream, while preserving the correct execution semantics of the program. It accomplishes this by reordering the uops to execute them as quickly as possible.

The out-of-order core will schedule for execution as many ready uops as possible each clock cycle, regardless of their original program order. By considering a larger number of uops from the program, the out-of-order core can usually find many independent uops that are ready to execute. The maximum number of uops that the out-of-order core can contain is 126, of which 48 can be load operations and 32 can be store operations.

At the heart of the out-of-order core are the uop schedulers. The schedulers determine when a uop is ready to execute by tracking its input register operands. When the input operands have been produced, the uop is considered to be ready to execute. The scheduler will then schedule the uop to execute when the execution resources required by the uop are available. Thus, uops are allowed to schedule and execute in what is called data-dependent order. In many code sequences, there are independent streams of execution. The scheduler identifies the streams of execution and allows these streams to execute in parallel with each other, regardless of their original program order.

There are five different schedulers connected to four different dispatch ports. On two of these ports, up to two uops can be dispatched each clock cycle. The fast Arithmetic and Logic Unit (ALU) schedulers can schedule on each half of a clock cycle, while the other schedulers can only schedule once per clock cycle. One fast ALU scheduler shares a dispatch port with the floating-point/media move scheduler, while the other fast ALU shares another dispatch port with the complex integer/complex floating-point/media scheduler. These schedulers arbitrate for a dispatch port when multiple schedulers have uops ready to execute at the same time. The remaining two dispatch ports allow one load and one store address uop to be dispatched every cycle. The collective dispatch bandwidth across all of the schedulers is six uops per clock cycle. This is twice the rate at which the out-of-order core can receive uops from the Execution Trace Cache and allows higher flexibility to issue ready uops on the different ports.

Rapid Execution Engine

The Rapid Execution Engine of the NetBurst microarchitecture executes up to six uops per main clock cycle. These uops are executed by several execution units: two double-speed integer ALUs, a complex integer unit, load and store Address Generation Units (AGUs), a complex floating-point/media unit, and a floating-point/media move unit. These highly tuned and optimized execution units are designed for low latency and high throughput.

The double-speed integer ALUs are able to execute at a rate of two uops per clock cycle, providing for a very high ALU throughput. Being able to execute these uops at twice the rate of the main core clock enables application performance to be increased relative to running the ALUs at the main clock rate.

The NetBurst microarchitecture is also able to execute one load and one store address uop every clock cycle through the AGUs. The AGUs are very tightly coupled to the low-latency first-level (L1) data cache. On this processor, the cache is 16 kilobytes in size and is used for both integer and floating-point/media loads and

stores. It is organized as an 8-way set associative write-through cache containing 64-byte cache lines.

The low latency of the L1 cache is very hard to achieve. This cache uses unique access algorithms to enable its low latency. The algorithms leverage the fact that almost all accesses hit the L1 data cache and the Data Translation Lookaside Buffer (DTLB). Generally, the schedulers assume that loads will hit the L1 data cache and will schedule dependent uops before the parent load has finished executing. Allowing these dependent uops to dispatch prior to knowing if the load has hit the cache is a form of data speculation. If the load misses the L1 data cache, the dependent uops will already be well into their execution and will temporarily be bound to incorrect data. Using a mechanism known as replay, the processor tracks and re-executes instructions that received incorrect data. Only the dependent operations are replayed; all independent operations are allowed to complete. Using this form of data speculation allows more parallel execution streams to be extracted from the program and increases the performance of the processor.

Floating-Point (x87), MMX, SSE (Streaming SIMD Extension), SSE2 (Streaming SIMD Extension 2), and the new SSE3 (Streaming SIMD Extension 3) operations are executed by the two floating-point execution blocks. One of the execution blocks is used for simple operations, such as SSE register-to-register moves and x87/MMX/SSE/SSE2 store data uops. The other execution block is used for more complex operations.

Store-to-Load Forwarding Enhancements

In all implementations of the NetBurst microarchitecture, stores are written to the L1 data cache in programmatic order and only after the store is guaranteed to be non-speculative. This requires that all operations older than the store must be completed before the store's data are committed to the cache. The forwarding mechanism implemented enables a load dependent on a store's data to have its data "forwarded" prior to the commitment of the store's data into the L1 cache. Forwarding is accomplished by doing a partial address match between the load and all older stores in the Store Forwarding Buffer (SFB) in parallel with the load's L1 data cache access. If the load's partial address matches that of an older store in the SFB, then the load gets its data from the SFB instead of the cache. The forwarding mechanism is optimized for speed such that it has the same latency as a cache lookup. To meet this latency requirement, the SFB cannot afford to do a full address and access size check. This function is accomplished by the Memory Ordering Buffer (MOB) later in the pipeline. The role of the MOB is to ensure that the forwarded load got the correct data from the most recent dependent store. In the event that the forwarding from the SFB was incorrect, the load in question must be re-executed after the dependent store writes to the L1 cache. The load can then pick up its data from the cache.

The latency from when a store has valid data to when these data are written into the cache can be high because of the deep pipeline of the NetBurst microarchitecture. So in cases where a load must wait for a store to commit its data for the load to complete, a significant reduction in performance can occur. Most of these cases are rare in real-world applications. However, there a few instances where applications do see a performance loss:

- Forwarding disabled due to address misalignment.

- Wrong forwarding due to a partial address match.

Mechanisms have been implemented on recent implementations of the Intel Pentium 4 processor to improve the performance in the above cases.

Force forwarding is a mechanism that allows the MOB to control the forwarding in the SFB. Figure 2 shows the block diagram for this mechanism. Two new selection points were added to the existing store-forwarding path. The forwarding-entry-selection mux allows the MOB to override the SFB's partial address match-based entry selection, while the data alignment mux allows for misaligned data to be rotated, based on the shift information provided by the MOB.

When a load first executes, the SFB detects a dependency with older stores based on a partial address match. When this load comes to the MOB to determine its "true" dependencies, the MOB can either agree with the SFB's decision to forward or it can cause the load to be re-executed. The load can be re-executed because the SFB detected either an incorrect dependency or because it failed to detect a dependency when a dependency did exist. If the SFB's dependency check is wrong, the MOB can correct the forwarding logic when the load re-executes by directing the SFB in one of two ways: forward to the load from the right entry and rotate the data as necessary or disable forwarding to the load if there is no dependent store in the SFB.
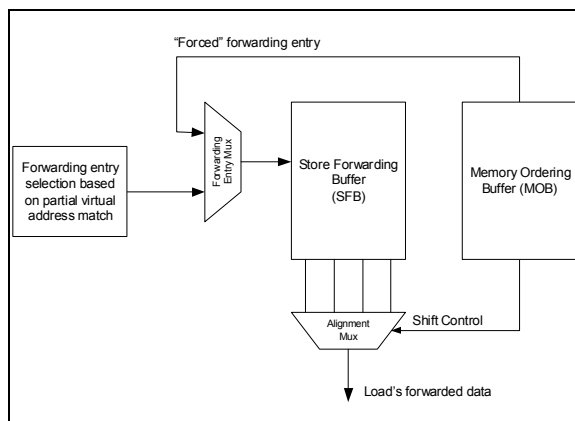
Figure 2: Force forwarding block diagram

The misaligned address cases that are fixed by the force forwarding mechanism are shown in Figure 3. In the figure, for each load at a given starting address, the data access sizes for which force forwarding is supported are listed. These cases can be categorized as follows:

- DWord/QWord Store forwarding to Byte/Word loads whose data are fully contained in either the lower or upper DWord.

- QWord Store forwarding to DWord Load to the upper DWord of the Store.

For each of these cases, the MOB "forces" the SFB to forward from a specific store by a given shift amount in order to align the store's data to the load.
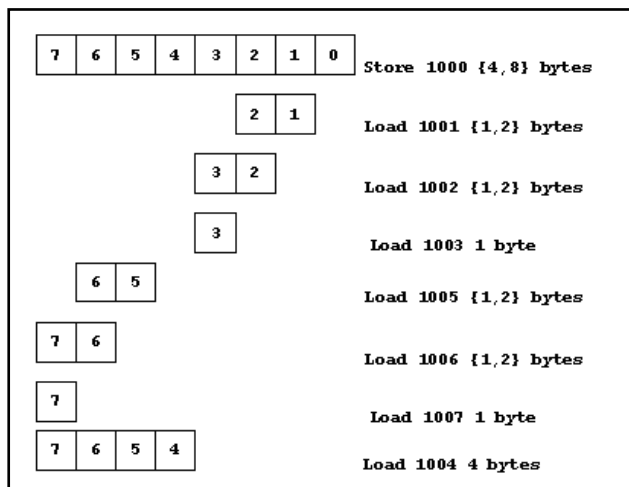

Figure 3: Supported cases of misaligned forwarding

False forwarding occurs when the SFB detects a partial address match between a load and a store, but their full addresses do not match. The MOB detects the false forward condition and determines if there exists another store that the load should have forwarded from. If a store exists that can be forwarded, then the MOB will direct the SFB to forward from this store entry using the

force forwarding mechanism when the load re-executes. If the MOB detects that there is no dependent store in the forwarding buffer, then the MOB instructs the SFB to not forward to this load. When the load is re-executed, it can then pick up its data from the cache instead.

NEW MICROARCHITECTURAL FEATURES AND ENHANCEMENTS

The 90nm Intel Pentium 4 processor improves performance over prior processor implementations through increasing the sizes of key resources, while also improving existing algorithms and introducing new microarchitectural features. These changes were made throughout the various parts of the processor as detailed below.

Front End

The instruction fetch and decode portions of this Intel Pentium 4 processor remain largely unchanged from previous implementations, but some performance enhancements have been made.

The simple static branch prediction scheme that is used when the Branch Target Buffer (BTB) has no prediction for a conditional branch has been enhanced. At the time the instruction decoder realizes that an instruction is a branch that was not predicted by the BTB, a static branch prediction is made. Making this prediction at decode time allows for a faster restart, and therefore better performance, rather than waiting for the normal execution time detection of a mispredicted branch.

In prior Pentium 4 processor implementations, the static prediction algorithm was to predict that a branch was taken if the branch direction was backwards and to predict that the branch was not taken if the branch jumps forward. This helped by correctly predicting *taken* for the first iteration of most loops. This works well for backwards branches that are in loops, but not all backwards branches are loop-ending branches.

We can try to ascertain the difference between loop-ending branches and other backwards branches by looking at the distance of the branch and the condition on which the branch is dependent. Our studies showed that a threshold exists for the distance between a backwards branch and its target; if the distance of the branch is larger than this threshold, the branch is unlikely to be a loop-ending branch. If the BTB has no prediction for a backwards branch, the Intel Pentium 4 processor will then predict *taken* for the branch only if the branch distance is less than this threshold.

We also discovered that branches with certain conditions were more often not taken, regardless of their

direction and distance. The conditions that they used are not common loop-ending conditions, so for branches with these conditions and no BTB prediction, the static prediction algorithm predicts them as *not taken*.

In addition to these changes in the static prediction algorithm, we also enhanced the dynamic branch prediction algorithms to reduce the number of times that a branch is mispredicted. Each time a branch is mispredicted, the pipeline must be flushed. Thus, large performance gains can be had by reducing the number of branch mispredictions. To this end, one of the dynamic branch predictor enhancements we made was to add an indirect branch predictor. This was motivated by results from the Intel Pentium® M processor team, who saw good performance improvements on some applications [3]. Table 1 compares the number of branch mispredictions per 100 instructions on the 90nm version of the Intel Pentium 4 processor versus the 130nm version of the processor on the components of SPECint*_base2000. The data were collected using the performance counters available on each processor, and they show the reduction in mispredictions on almost all components, due to the algorithmic enhancements.

Table 1: Comparison of mispredicted branches per 100 instructions

	130nm	90nm
164.gzip	1.03	1.01
175.vpr	1.32	1.21
176.gcc	0.85	0.70
181.mcf	1.35	1.22
186.crafty	0.72	0.69
197.parser	1.06	0.87
252.eon	0.44	0.39
253.perlbmk	0.62	0.28
254.gap	0.33	0.24
255.vortex	0.08	0.09
256.bzip2	1.19	1.12
300.twolf	1.32	1.23

Another performance enhancement was to expand the set of instructions where the processor detects that dependence chains can be broken. A common technique to zero a register is to `xor` the register with itself, rather than to move an immediate of 0 into the register. This technique is preferred because of the smaller resulting code size. The result is logically equivalent, but the `xor` method adds a dependency on the previous contents of the register. In an out-of-order machine, this extra dependency can result in a performance loss. Previous processor implementations recognized when the `xor`, `pxor`, and `sub` instructions were used in this manner, and they removed the dependency on the source register, since the same answer is arrived at regardless of the value of the sources. On this Intel Pentium 4 processor, additional instructions that are used for the same purpose are now detected. Among these are the SSE instruction `xorps` and the SSE2 `psub` and `xorpd` instructions.

We can also now encode more types of uops inside the trace cache than could be encoded in prior processors. If an instruction uses a uop that cannot be encoded in the trace cache, then the uops for the entire instruction have to be sequenced from the Microcode ROM. This enhancement allows for higher average uop bandwidth from the front end of the machine to the execution core by removing transitions to the Microcode ROM. Indirect calls with a register source operand and software prefetch instructions are the best examples of instructions that can now be encoded in the trace cache.

Execution Core

The execution core of the Intel Pentium 4 processor is similar to previous implementations in that the two integer ALUs run at 2x the frequency of the rest of the

---

® Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

processor, allowing for high throughput of common arithmetic and logical operations. An enhancement we implemented in this processor was to add a shifter/rotator block to one of the ALUs. This block allows the most common forms of shift and rotate instructions to be executed on a fast ALU. On prior Pentium 4 processor implementations, these operations were executed as complex integer operations that took multiple cycles to execute.

Another key operation whose latency has been reduced on this processor is integer multiply. Previously, the Intel Pentium 4 processor executed integer multiplies using the floating-point multiplier. This introduced latency by paying the cost of moving the source operands to the floating-point side and then moving the result back to the integer side. On this processor, we added a dedicated integer multiplier to service these operations.

On top of the changes to the execution units, we also changed the L1 data cache. As with all implementations of the NetBurst microarchitecture, the cache is designed to minimize the load-to-use latency by using a partial virtual address match to detect early in the pipeline whether a load is likely to hit or miss in the cache. On this processor, we significantly increased the size of the partial address match from previous implementations, thus reducing the number of false aliasing cases. More importantly, we increased the size of the cache. Previously, the L1 data cache was 8 kilobytes in size and 4-way associative. Now the size of the cache has been increased to 16 kilobytes by increasing the associativity to 8-ways.

The schedulers in the NetBurst microarchitecture are critical, as they must run at a high speed in order to continually feed the high-speed execution core. The schedulers in this implementation of the microarchitecture remain largely the same, as the rate at which they can feed the core is unchanged from prior implementations. In all implementations, the schedulers are capable of scheduling up to six uops per clock cycle.

Even though the rate of scheduling remains the same, we made several enhancements to the schedulers to improve performance on the implementation. The two schedulers that are used to hold uops used in x87/SSE/SSE2/SSE3 instructions were increased in size. By increasing the size of these schedulers, the window of opportunity to find parallelism in multimedia algorithms is increased. And we increased the effective size of the queues that feed all the schedulers, such that more uops can now be buffered between the allocator and the scheduler before the allocator has to stall. This allows the allocation and renaming logic to continue to

look ahead in the instruction stream even when the schedulers are full.

Additionally, we changed the mechanism used to schedule load uops to improve performance. As on prior implementations, store instructions are broken up into two pieces: a store address and a store data uop. In the previous implementations, loads were scheduled asynchronously to store data uops. Thus, if a load needed to receive forwarded data from a store, it was possible that the load would execute before the store data uop. If this occurred, the load would have to be re-executed after the store data uop had finally executed. Because of this, latency could be introduced because the minimum latency between a store data uop and a dependent load was not the common case latency for loads that had been re-executed. On top of that penalty, having to re-execute the load meant that precious load bandwidth was being wasted on loads that executed more than once. To alleviate both of these issues, we added a simple predictor to the processor that marks whether specific load uops are likely to receive forwarded data, and, if so, from which store they are likely to forward. Given this information, the load scheduler now holds a load that is predicted to forward in the scheduler until the store data uop that produces the data it depends on is scheduled. In doing so, both of these performance penalties are reduced significantly.

We also added a performance feature to enhance applications that use the SSE/SSE2/SSE3 instructions. On the x87 side, the Floating-Point Control Word (FCW) is often modified as the programmer wants to change the rounding mode and precision of the data that are being worked with. To avoid serializing the processor each time that the FCW is modified, a simple prediction scheme was implemented on the NetBurst microarchitecture to capture common renaming cases. This same idea is now extended on this implementation of the microarchitecture to also handle the MXCSR, which is the corollary of the FCW for instructions that use the SSE registers. On prior implementations, changes to the MXCSR would serialize the machine. On this processor, the common case modifications of MXCSR will not incur a serialization.

Memory System

In the memory subsystem of the processor, we made a number of changes to increase overall performance. The changes made focus on trying to reduce the amount of time spent waiting for data to be fetched from DRAM and on increasing the size of critical resources so as to limit the number of times the processor is forced to stall because of a resource shortfall.

One mechanism to reduce the amount of time spent waiting for data to be returned from DRAM is to increase the size of the caches. Previous implementations of the Intel Pentium 4 processor contained unified L2 caches of either 256 or 512 kilobytes. On the 90nm version of the Intel Pentium 4, we implemented a 1MB L2 unified cache. Similar to the previous implementations, the cache is a writeback 8-way set associative cache and contains 128-byte lines.

A second way to reduce the time waiting for DRAM is by using software prefetch instructions that are inserted by the programmer to bring data into the cache before the data are actually used. On all Pentium 4 processors, software prefetch instructions bring in data from DRAM into the L2 cache. These instructions opportunistically look up the L2 cache and on a miss, initiate a data prefetch cycle on the front-side bus. The data are filled only to the L2 cache so as not to pollute the much smaller L1 data cache.

On previous Pentium 4 processor implementations, these operations were dropped on a DTLB miss. The Pentium 4 processor adds a mechanism to allow the software prefetch instructions to initiate page table walks and allow data TLB fills if the prefetch access is to a page currently not cached in the TLB. We added special fault-handling logic to handle cases where page faults were detected on the software prefetch instructions. These instructions are dropped silently without reporting the fault to the operating system, and the prefetch operation is not performed. In effect, the 90nm version of the Pentium 4 processor allows software prefetch instructions to not only prefetch data, but also to prefetch page table entries into the DTLB. As we previously mentioned, the cost of software prefetch instructions has been greatly reduced on this processor, as software prefetches can now be cached in the trace cache; they used to have to be fetched from the Microcode ROM.

A third mechanism used to reduce the time waiting for DRAM is through a hardware prefetching scheme. The hardware prefetcher looks for streams of data and tries to predict what data will be needed next by the processor and proactively tries to fetch these data. This mechanism can be superior to software prefetching, as it requires no effort from the programmer and can improve performance on code that has no software prefetch instructions. All Intel Pentium 4 processors contain a hardware prefetcher that can prefetch both code and data streams, where the data stream can be accessed by loads and/or stores. This implementation of the processor improves upon the previous implementations in its ability to detect when to prefetch data and what data needs to be prefetched. Figure 4 shows the effect of the

hardware prefetcher. We show the performance of this processor with the hardware prefetcher enabled versus the hardware prefetcher disabled on the most hardware prefetcher-sensitive components in the SPECint_base2000 and SPECfp*_base2000 benchmarks[1]. These are the components that gain more than 10% in performance by enabling the hardware prefetcher.

Addressing resource constraints was the other means of improving performance in the memory system. On previous Intel Pentium 4 processors, only 24 stores could be simultaneously outstanding in the processor. This number has now been increased to 32. Additionally, the number of write-combining buffers that are used to track streams of stores was increased from 6 to 8, which also alleviates pressure on the number of stores that can be in the machine simultaneously by allowing stores to be processed faster. Finally, the number of unique outstanding loads that have missed the L1 data cache and can be serviced has been increased from 4 to 8.
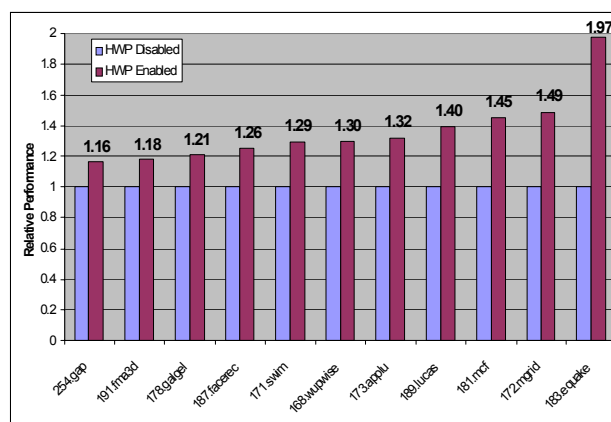

Figure 4: Effect of the hardware prefetcher

HYPER-THREADING TECHNOLOGY

Hyper-Threading Technology was introduced on previous implementations of the Intel Pentium 4 processor and is also present on many versions of this latest processor. Hyper-Threading Technology allows one physical processor to appear to the operating system as two logical processors [2]. This allows two program software threads, either related or unrelated, to execute

---

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

[1] Estimated performance through measurements on non-production hardware.

simultaneously throughout the processor. Prior to Hyper-Threading Technology, only one thread could be executed at a time on a processor, and each switch to a different thread would incur a context-switching overhead penalty.

Many of the changes mentioned previously were motivated mainly by Hyper-Threading Technology performance. For instance, increasing the number of outstanding loads that miss the L1 data cache from 4 to 8 has very little performance impact on the majority of single-threaded applications. This resource, however, is more important when two threads are being executed. Increasing the size of the resource that controls this behavior provides for better threaded performance while also slightly enhancing single-threaded performance. Similarly, the size of the queue that sits between the front end of the processor and the allocation/rename logic was also increased in this processor implementation. Again this change was motivated by the need for increased performance when running multiple threads, as the size increase provides minimal benefit when only running a single thread.

Other changes that were made in this processor implementation to help support Hyper-Threading Technology performance include additions to the type of operations that can be conducted in parallel. For instance, on previous implementations, the processor could either work on a page table walk or on handling a memory access that splits a cache line, but not on both simultaneously. For single-thread performance, this limitation was rarely seen as a bottleneck. However, when running multiple threads, the effect of this bottleneck becomes much more acute as the behavior of one thread can have a significant negative impact on the other thread. In this processor, this bottleneck has been fixed such that a page table walk can occur at the same time as a memory access that splits a cache line is being handled. Similarly, on prior implementations, if a page table walk missed all the caches and had to go to DRAM, no new page table walks could be started. This again was very rarely seen as a bottleneck for single-threaded performance but was detrimental when running multiple threads as one poorly behaving thread could effectively stall both threads. Now, in this implementation, a page table walk that misses all of the caches and goes to DRAM does not block other page table walks from being initiated.

Changes were also made to some of the thread selection points in this version of the Pentium 4 processor in order to improve overall bandwidth. For example, the trace cache now responds faster to stalling events in the core, dedicating all of its resources to the thread that is not stalled, thereby generating better overall performance.

In addition to these changes, this processor also contains an enhancement for Hyper-Threading Technology performance known as the context identifier that was included in some prior processor implementations. With Hyper-Threading Technology, the partial virtual address indexing scheme used for the L1 cache creates conflicts when each logical processor's access pattern matches the partial virtual tag even when accessing separate regions of physical memory. For example, this situation can occur if the stacks of the two threads are offset by a fixed amount that is greater than the size of the partial match, such that these two addresses, although different, alias to the same partial tag. This causes contention in the cache, leading to a reduced cache hit rate. In order to reduce the likelihood of contention, a context identifier bit is added to the partial virtual tag for each logical processor. This bit is dynamically set or reset based on the page-table structure initialization for each logical processor and serves as an indication of data sharing intent across logical processors.

For example, assume that two logical processors share the same page directory base in physical memory. This gives a strong indication that data are intended to be shared between the logical processors. In such a case, the additional context-identifier bit for each logical processor is set to the same value, allowing for sharing of the L1 data cache. Conversely, if the page-directory bases are different, it is likely that both logical processors are working on separate data regions. In such a case, sharing of the L1 data cache is disallowed by keeping the context-identifier bit different across logical processors.

There may be uncommon cases where logical processors use different page directory bases but still share the same physical memory region through page-table aliasing. These arise when two different page table entries across logical processors point to the same physical page frame. The processor detects such cases and implements a reservation mechanism to prevent repetitive L1 cache access conflicts among different logical processors.

SSE3 INSTRUCTIONS

The Intel Pentium 4 processor extends the IA-32 ISA with a set of 13 new instructions. With the exception of three (`fisttp`, `monitor`, `mwait`), these instructions use the SSE registers. These new instructions are designed to improve performance in the following areas:

- x87 to integer conversion (`fisttp`)

- Complex arithmetic (`addsubps`, `addsubpd`, `movsldup`, `movshdup`, `movddup`)

- Video encoding (`lddqu`)

- Graphics (`haddps`, `hsubps`, `haddpd`, `hsubpd`)

- Thread synchronization (`monitor`, `mwait`)

Improved x87 Conversions to Integer

`Fisttp` has been added to provide the ability of IA-32 to ignore the value of the Floating-Point Control Word (FCW) when converting a value from x87 to an integer. Currently on IA-32, a conversion to integer is done with the convert-store instruction `fistp`. The rounding mode used for the conversion is taken from the FCW. In order to meet Fortran and C/C++'s specifications for conversion to integer, the rounding mode has to be set to *chop*, whereas the default rounding mode is usually set to *even* to minimize rounding errors. Because `fistp` gets its rounding mode from FCW, the user has to create a new FCW that is equal to the default one, but with the rounding mode changed to chop. Once FCW is changed, `fistp` can be used to do the conversion. Finally, the user has to restore the default value of FCW. The whole operation involves changing FCW twice, and since `fldcw` is a relatively slow instruction, it can degrade the performance of an application. To alleviate this problem, `fisttp` has been added. It is a new `fistp` instruction that ignores FCW and always uses chop as its rounding mode.

As shown below, the benefit of `fisttp` is two-fold: fewer instructions are needed and there is no need to modify FCW. The instruction is available in three precisions: Word (16-bit), DWord (32-bit), and QWord (64-bit).

Code without `SSE3`:
```
        fstcw <old FCW>
        movw ax, <old FCW>
        or ax, 0xc00
        movw <new FCW>, ax
        fldcw <new FCW>
        fistp <INT>
        fldcw <old FCW>
```

Code with `SSE3`:
```
        fisttp  <INT>
```

Complex Arithmetic

Complex arithmetic usage is ubiquitous, as it is used in Discrete/Fast Fourier Transform (DFT/FFT), Discrete Multi Tone (DMT) modulators, frequency domain filtering, etc. A typical example of the importance of complex arithmetic in a multimedia context is given by the implementation of an Acoustic Echo Canceller (AEC). In an AEC, a long Finite Impulse Response (FIR) filter is used to model the inverse of the acoustic channel. It is not uncommon for this filter to have 1024 or more taps. The operation done by a FIR filter is called a convolution, and its execution time is $O(n^2)$. With filters of such large length, and with the quadratic cost of a convolution, the operation of filtering in the time domain can be prohibitive, to the point of not meeting, for example, a real-time constraint. By moving from the time domain to the frequency domain, the execution time can be significantly reduced. Because the execution time of a DFT is also $O(n^2)$, moving to the frequency domain does not appear to have saved anything. But DFT has fast implementations with execution time $O(nlogn)$. Such fast implementations of DFT are collectively called FFT. In the frequency domain, a convolution ($O(n^2)$) is simply a point-product ($O(n)$). For a filter with fixed coefficients, the n-element input array can be transformed into the frequency domain in $O(nlogn)$ operations; the point-multiplication (with the frequency domain transformed set of coefficients) takes $O(n)$ operations; the conversion of the result back to the time domain (using an inverse FFT) takes also $O(nlogn)$ operations. For large n, the complexity behaves as $O(nlogn)$, significantly faster than $O(n^2)$.

Three benchmarks out of SPEC[*] CPU2000[*] make heavy use of complex arithmetic: 168.wupwise (BLAS3 ZGEMM – complex matrix multiply), 189.lucas (FFT_SQUARE – a FFT-based function to square large integer numbers), and 187.facerec (FFT).

Five instructions have been added to significantly accelerate complex arithmetic. Two instructions (`addsubps` and `addsubpd`) perform a mix of floating-point addition and subtraction, hence removing the need for changing the sign of some operands. The

three others (`movsldup`, `movshdup`, `movddup`), in their memory version, combine loads with some level of duplication, hence saving the need for a shuffle instruction on the loaded data.

Code without SSE3:

```
movapd    xmm0, <mem_X>
movapd    xmm1, <mem_Y>
movapd    xmm2, <mem_Y>
unpcklpd  xmm1, xmm1
unpckhpd  xmm2, xmm2
mulpd     xmm1, xmm0
mulpd     xmm2, xmm0
xorpd     xmm2, xmm7
shufpd    xmm2, xmm2, 0x1
addpd     xmm2, xmm1
movapd    <mem_Z>, xmm2
```

Code with SSE3:

```
movapd    xmm0, <mem_X>
movddup   xmm1, <mem_Y>
movddup   xmm2, <mem_Y+8>
mulpd     xmm1, xmm0
mulpd     xmm2, xmm0
shufpd    xmm2, xmm2, 0x1
addsubpd  xmm2, xmm1
movapd    <mem_Z>, xmm2
```

The code sequence above shows how to implement a double-precision complex multiplication using SSE2 only or with the new SSE3 instructions, where mem_X contains one complex operand and mem_Y the other; mem_Z is used to store the complex result; and xmm7 is a constant used to change the sign of one data element. Since the main speed limiter of this code is the number of execution uops (7 for SSE2, 4 for SSE3), the new instructions can improve complex multiplications by up to 75%. On SPEC CPU2000, the compiler is able to use SSE3 to improve 168.wupwise by 10-15%.

Video Encoding

The most compute-intensive part of a video encoder is usually Motion Estimation (ME) where blocks from the current frame are checked against blocks from the previous frame to find the best match. Many metrics can be used to define the best match. The most common is the L1 metric: the sum of absolute differences. Due to the nature of ME, loads of the blocks from the previous frame are unaligned whereas loads of the blocks from the current frame are aligned. Unaligned loads suffer two penalties:

- cost of handling the unaligned access

- impact of the cache line splits

The NetBurst microarchitecture does not support a uop to load 128-bit unaligned data. For that reason, 128-bit unaligned load instructions, such as `movups` and `movdqu`, are emulated with microcode, using two 64-bit loads whose results are merged to form the 128-bit result. In addition to the cost of the emulation, unaligned loads are penalized by the cost of handling cache line splits if the access crosses a 64-byte boundary.

SSE3 adds `lddqu` to solve the cache line split problem on 128-bit unaligned loads. The instruction works by loading a 32-byte block aligned on a 16-byte boundary, extracting the 16 bytes corresponding to the unaligned access. Because the instruction loads more bytes than requested, some usage restrictions apply. `Lddqu` should be avoided on Uncached (UC) and Write-Combining (USWC) memory regions. Also, by its implementation, `lddqu` should be avoided in situations where store-load forwarding is expected. In load-only situations, and with memory regions that are not UC or USWC, `lddqu` can advantageously replace `movdqu`/`movups`/`movupd`.

The code below shows an example of using the new instruction. Both code sequences are similar except that the load unaligned (`movdqu`) is replaced by the new unaligned load (`lddqu`). With the assumption that 25% of the unaligned loads are across a cache line, the new instruction can improve the performance of ME by up to 30%. MPEG* 4 encoders have demonstrated speedups greater than 10%.

Motion Estimator without `SSE3`:

```
movdqa xmm0, <current>
movdqu xmm1, <previous>
psadbw xmm0, xmm1
paddw  xmm2, xmm0
```

Motion Estimator with `SSE3`:

```
movdqa xmm0, <current>
lddqu  xmm1, <previous>
psadbw xmm0, xmm1
paddw  xmm2, xmm0
```

Graphics

Most (graphics) vertex databases are organized as an array of structures (AOS), where each vertex structure contains data fields such as the following:

- x, y, z, w: coordinates of the vertex

- n_x, n_y, n_z, n_w: coordinates of the normal at the vertex

- r, g, b, a: colors at the vertex

- u0, v0: 1st set of 2D texture coordinates

- u1, v1: 2nd set of 2D texture coordinates

---

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

By its very nature, SSE does not deliver optimal results when operating on vertex databases organized as an AOS. SSE is much better at handling vertex databases organized as a structure of arrays (SOA), where the first array contains the x of all the vertices; the second array, the y of all the vertices; etc. Because AOS is the favored way vertex databases are organized, in order to use SSE, the data have to be loaded and reorganized using shuffle instructions.

The most common operation performed in a vertex shader is the scalar product, where 3 (or 4) pairs of single-precision data elements are multiplied and the 3 (or 4) results summed. Due to the AOS organization of the vertex database, evaluating the scalar product can be challenging with SSE because of the lack of horizontal instructions. We have added horizontal floating-point addition/subtraction instructions to speed up the evaluation of scalar products.

The code sequence below illustrates how a scalar product of four single-precision pairs of elements can be evaluated with and without the new instructions:

Code without SSE3:

```
mulps   xmm0, xmm1
movaps  xmm1, xmm0
shufps  xmm0, xmm1, 0xb1
addps   xmm0, xmm1
movaps  xmm1, xmm0
shufps  xmm0, xmm0, 0x0a
addps   xmm0, xmm1
```

Code with SSE3:
```
mulps    xmm0, xmm1
haddps   xmm0, xmm0
haddps   xmm0, xmm0
```

Thread Synchronization

Monitor and mwait instructions provide a solution to address Hyper-Threading Technology performance of the operating system idle loop and other spin-wait loops in operating systems and device drivers. Software can use the monitor and mwait instructions to hint that a thread is not doing useful work (e.g., spinning and waiting for work). The processor may then go into a low-power and performance-optimized state. Monitor and mwait provide a way for software to wake up the processor from this low-power/performance-optimized state via a store to a specified memory location (e.g., a store to the work queue).

Monitor sets up hardware to detect stores to an address range, generally a cache line. The monitor instruction relies on a state in the processor called the *monitor event pending flag*. The monitor event pending flag is either set or clear and its value is not architecturally visible except through the behavior of the mwait instruction. The monitor event pending flag is set by multiple events including a write to the address range being monitored and reset by the monitor instruction.

The monitor instruction sets up the address monitoring hardware using the address specified in EAX and resets the monitor event pending flag. A store to the address range will set the monitor event pending flag. Other events will also set the monitor event pending flag, including interrupts or any event that may change the page tables. The content of ECX and EDX are used to communicate other information to the monitor instruction.

Mwait puts the processor into the special low-power/optimized state until a store, to any byte in the address range being monitored, is detected, or if there is an interrupt, exception, or fault that needs to be handled. There may also be other time-outs or implementation-dependent conditions that may cause the processor to exit the optimized state. The mwait instruction is architecturally identical to a nop instruction. It is effectively a hint to the processor to indicate that the processor may choose to enter an implementation-dependent optimized state while waiting for an event or for a store to the address range set up by the preceding monitor instruction in program flow. For example, a Hyper-Threading Technology-capable processor may enter a state that allows the other thread to execute faster, or it may enter a state that allows for lower power consumption, or both.

The monitor and mwait instructions must be coded in the same loop because execution of the mwait instruction will clear the monitor address range. It is not

possible to execute `monitor` once and then execute `mwait` in a loop. Setting up `monitor` without executing `mwait` has no adverse effects.

Typically the monitor/mwait pair is used in a sequence like this:

```
EAX = Logical Address(Trigger)
ECX = EDX = 0                          // Hints
While ( !trigger_store_happened) {
        MONITOR EAX, ECX, EDX
        If ( !trigger_store_happened ) {
                MWAIT EAX, ECX
        }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the `monitor` instruction. Without the second check that triggering store would go un-noticed.

It is expected that operating systems will use the `monitor` and `mwait` instructions to significantly improve the performance of idle loop handling and allow the system to provide higher performance at lower power consumption.

PERFORMANCE

Given all of these changes in the 90nm version of the Intel Pentium 4 processor, the real question is how much performance benefit will be realized on applications from making these changes.

Table 2: Detailed system configuration for results shown in Figure 5

Processor	Pre-production Intel® Pentium® 4 processor 3.40 GHz supporting Hyper-Threading Technology	Pre-production Intel® Pentium® 4 processor 3.40E GHz supporting Hyper-Threading Technology
Motherboard	Intel Desktop Board D875PBZ AA-204	Intel Desktop Board
Motherboard BIOS	BZ87510A.86A.0041.P09	Pre-production BIOS
Cache	512KB full-speed Advanced Transfer Cache	1MB full-speed Advanced Transfer Cache
Memory Size	1 GB (2x512MB) PC3200 DDR400 (Samsung* PC3200U-30331-B2 M368L6423ETM-CCC CL3 Double-Sided DDR400 memory)	1 GB (2x512MB) PC3200 DDR400 (Samsung* PC3200U-30331-B2 M368L6423ETM-CCC CL3 Double-Sided DDR400 memory)
Hard Disk	Seagate* ST3160023AS 160 GB Serial ATA (SATA) (7200 RPM, 8MB cache)	Seagate* ST3160023AS 160 GB Serial ATA (SATA) (7200 RPM, 8MB cache)
Hard Disk Driver	Intel Application Accelerator RAID Edition 3.5 with RAID ready	Intel Application Accelerator RAID Edition 3.5 with RAID ready
Video Controller/Bus	ATI* Radeon* 9800 Pro 8x AGP	ATI* Radeon* 9800 Pro 8x AGP
Video Memory	128 MB DDRAM	128 MB DDRAM
Operating System	Microsoft* Windows* XP Professional, Build 2600, Service pack 1 on NTFS Default Microsoft DirectX* 9.0b	Microsoft* Windows* XP Professional, Build 2600, Service pack 1 on NTFS Default Microsoft DirectX* 9.0b
Video Driver Revision	ATI Catalyst* 3.5 Driver Suite: display driver version: 6.14.10.6360	ATI Catalyst* 3.5 Driver Suite: display driver version: 6.14.10.6360
Graphics	1024x768 resolution, 32-bit color	1024x768 resolution, 32-bit color
SPEC* CINT2000	Intel C++ Compiler Plug-in V8.0 Microsoft Visual Studio* .NET V7.0 (for libraries)	Intel C++ Compiler Plug-in V8.0 Microsoft Visual Studio* .NET V7.0 (for libraries)
SPEC* CFP2000	Intel C++ Compiler Plug-in V8.0 and Intel FORTRAN Compiler Plug-in V8.0 Microsoft Visual Studio .NET V7.0 (for libraries)	Intel C++ Compiler Plug-in V8.0 and Intel FORTRAN Compiler Plug-in V8.0 Microsoft Visual Studio .NET V7.0 (for libraries)

Table 3: Detailed system configuration for results in Figure 6

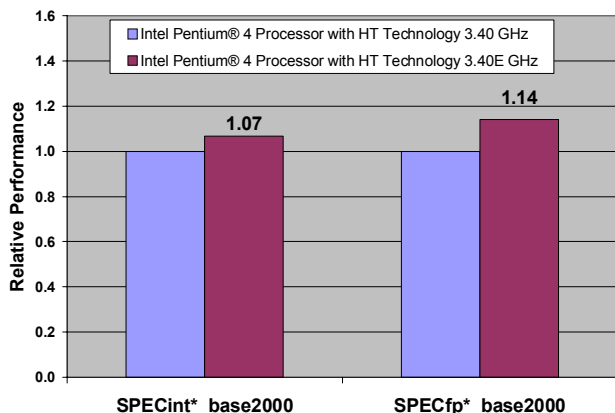| Processor	Pre-production Intel® Pentium® 4 processor 3.40E GHz supporting Hyper-Threading Technology
Motherboard	Intel Desktop Board
Motherboard BIOS	Pre-production BIOS
Cache	1MB full-speed Advanced Transfer Cache
Memory Size	512MB (4x128MB) Samsung PC3200U-30330-C3 M368L1624DTM-CCC 128MB DDR PC3200 CL3 Single-Sided DDR400 memory
Hard Disk	IBM 120GXP 80 GB IC35L080AVVA07-0 ATA-100
Hard Disk Driver	MS default UDMA-5
Video Controller/Bus	ATI Radeon 9700 Pro AGP graphcis
Video Memory	128 MB DDRAM
Operating System	Microsoft* Windows* XP Professional, Build 2600, Service pack 1 on NTFS Default Microsoft DirectX* 9.0b
Video Driver Revision	ATI CATALYST 6.13.10.6166 driver
Graphics	1024x768 resolution, 32-bit color

Figure 5: Performance comparison (estimated SPEC* CPU2000* performance as measured on pre-production hardware)

Figure 5 compares the performance of this new processor with the performance of the 130nm version of the Intel Pentium 4 processor with a 512kb L2 cache on SPEC CPU2000, as estimated on pre-production hardware. Detailed system configuration information is shown in Table 2. As can be seen here, the performance enhancements that have been described in this paper do have a noticeable effect on overall performance.
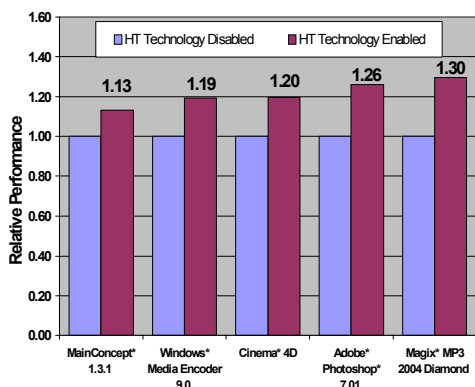

Figure 6: Performance benefit of Hyper-Threading Technology

Hyper-Threading Technology on this processor also shows significant benefits on popular consumer applications and for various multi-tasking scenarios. Figure 6 compares the performance on some of these applications and scenarios when Hyper-Threading Technology is enabled and disabled on this processor.

Table 3 lists the detailed system configuration for these results.

CONCLUSION

The NetBurst microarchitecture that was introduced with the Intel Pentium 4 processor brought unprecedented levels of performance to the end user through its unique features such as the Execution Trace Cache and an execution core that ran at 2x the core frequency. Now, we are building upon the strength of those previous processors with the new Intel Pentium 4 processor manufactured on the 90nm process. With these new performance features and enhancements, the performance of desktop processors continues to reach new heights. With capabilities like Hyper-Threading Technology and a set of new instructions, building blocks are being provided for software to be created to take advantage of this power and deliver to users a new level of functionality on their desktop.

ACKNOWLEDGMENTS

REFERENCES

[1]. Hinton, G.; Sager, D.; Upton, M.; Boggs, D.; Carmean, D.; Kyker, A.; Roussel, P., "The Microarchitecture of the Pentium® 4 Processor," *Intel Technology Journal* Q1, 2001.

[2]. Marr, D.; Binns, F.; Hill, D.; Hinton, G.; Koufaty, D.; Miller, J.; Upton, M., "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History," *Intel Technology Journal*, Q1, 2002. http://developer.intel.com/technology/itj/2002/volume06issue01/

[3]. Gochman, S.; Ronen, R.; Anati, I.; Berkovits, A.; Kurts, T.; Naveh, A.; Saeed, A.; Sperber, Z.; and Valentine, R., "The Intel® Pentium® M Processor: Microarchitecture and Performance," *Intel Technology Journal*, Q2, 2003. http://developer.intel.com/technology/itj/2003/volume07issue02/

AUTHORS' BIOGRAPHIES

Darrell Boggs is a senior principal engineer/architect with Intel Corporation and has been working as a microarchitect for 12 years. He graduated from Brigham Young University with a M.S. degree in Electrical Engineering. Darrell played a key role on the Intel® Pentium® Pro processor design, and was one of the key

---

® Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

architects of the Pentium 4 processor. Darrell holds many patents in the areas of register renaming; instruction decoding; events and state recovery mechanisms; speculative architectures; and Hyper-Threading Technology. His e-mail address is darrell.boggs at intel.com.

Aravindh Baktha has been with Intel for 12 years. He worked on the design and microarchitecture of the Pentium 4 processor. Prior to joining the Pentium 4 processor team, Aravindh worked on the design of the 80960HA processor in Arizona and the Itanium® processor in California. Aravindh received his undergraduate degree from the University of Zambia and his M.S. degree in Electrical and Computer Engineering from Illinois Institute of Technology. His e-mail address is aravindh.baktha at intel.com

Jason M. Hawkins received his B.S. degree in Electrical and Computer Engineering from Brigham Young University. He joined Intel in 1997 and has focused on validation and microarchitecture of the Pentium 4 family of processors. His e-mail address is jason.hawkins at intel.com.

Deborah T. Marr is the CPU architect responsible for Hyper-Threading Technology in the Desktop Products Group. Deborah has been at Intel for over thirteen years. She first joined Intel in 1988 and made significant contributions to the Intel 386SX processor, the Pentium Pro processor, and the Pentium 4 processor. Her interests are in high-performance microarchitecture and performance analysis. Deborah received her B.S. degree in EECS from the University of California at Berkeley in 1988 and her M.S. degree in ECE from Cornell University in 1992. Her e-mail address is debbie.marr at intel.com.

John (Alan) Miller has worked at Intel for over seven years. During that time, he has worked on design and architecture for the Pentium 4 processor. Alan obtained his M.S. degree in Electrical and Computer Engineering from Carnegie Mellon University. His e-mail address is alan.miller at intel.com.

Patrice Roussel graduated from the University of Rennes in 1980 and L'Ecole Supérieure d'Electricité 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 microprocessor. In late 1991, he moved to Intel in Oregon to work on the Pentium Pro processor. Since 1995, he has been the floating-point architect of the Pentium 4 processor. His e-mail address is patrice.roussel at intel.com.

Ronak Singhal received his B.S. and M.S. degrees in Electrical and Computer Engineering from Carnegie Mellon University. He subsequently joined Intel in 1997 and has spent the majority of his time focused on microarchitecture performance analysis and verification for the Pentium 4 processors. His e-mail address is ronak.singhal at intel.com.

Bret Toll received his B.S. degree in Electrical Engineering from Portland State University and M.S. degree in Computer Science and Engineering from Oregon Graduate Institute. He joined Intel in 1993 and has focused on microcode, machine check architecture, and instruction decode microarchitecture for the Pentium 4 processor. In his spare time he likes to tinker with cars and is currently building a 1965 Ford roadster from a kit of components and hand-picked items from junkyards. His e-mail address is bret.toll at intel.com.

K. S. Venkatraman received his B.S. degree from Birla Institute of Technology and M.S. degree from Villanova University. He joined Intel in 1997 and has focused on microarchitecture as well as post-silicon performance analysis for the Pentium 4 processor. In his spare time, he enjoys amateur radio and riding his motorcycle. His e-mail address is k.s.venkatraman at intel.com.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm