Performance Characterization of Intel's Internet Streaming SIMD Extensions

by

Vikram Uday Godbole

Technical Report

UT ECE Tech Report 05062000

The University of Texas at Austin

May 2000

Acknowledgements

The author would like to thank Dr Lizy John for her advice and support during this project. We also appreciate the support given by Dell Computer Corp. through the Dell LARIAT program that funded this research. Thanks are due to Derek Soeder who pulled me out of a hole during the development of one of the tools. Finally, I would like to thank all members of the Laboratory for Computer Architecture, especially Room Five-Oh-Niners [thanks, J-Man! :-)], who helped me get my thoughts in place.

May 5, 2000.

Performance Characterization of Intel's Internet Streaming SIMD Extensions

by

Vikram Uday Godbole, M.S.E.
The University of Texas at Austin, 2000
SUPERVISOR: Lizy Kurian John

Over the past few years, the proliferation of compute-intensive multimedia applications has led several microprocessor vendors to incorporate SIMD extensions into the base ISAs. The trend began with the introduction of the MMX extensions on Intel's Pentium-series processors in 1996, followed by AMD's 3DNow! Architecture on the K6-2 and K7 processors. Latest in this line are the Internet Streaming SIMD Extensions (the ISSE, or more simply, the SSE) introduced on Intel's Pentium III processor. Intended to address the needs of the increasingly important, floating-point intensive, 'visual computing' applications, the SSE provide acceleration in the form of *floating point* SIMD operations.

This project examines some SSE-enhanced applications, and characterizes their performance on a Pentium III system. We also examine the speedup obtained by using prefetch instructions on a set of kernels. Finally, we compare performance of a popular 3D graphics benchmark on a Pentium II system with that on a Pentium III system. Evaluation has been done using a set of custom tools developed for that purpose. Analysis of the results has shown that performance on kernels is markedly improved over that on non-enhanced versions; performance improvement over applications is more modest.

Table of Contents

Chapter 1 : Introduction

The quest for audio and visual realism in contemporary multimedia applications has placed a heavy demand on the pixel processing power and memory bandwidth of today's computer systems, and multimedia applications have emerged as important computing workloads [2]. In response to this need for greater processing power, extensions to the base instruction set architectures have been introduced in many general-purpose processors. The Visual Instruction Set (VIS) on Sun's UltraSPARC, the MMX and SSE extensions on Intel Architecture Processors, AMD's 3DNow! [7] and Motorola's AltiVec Technology [6] are examples. These extensions enhance performance on the highly vectorizable media-processing algorithms through Single Instruction Multiple Data (SIMD) techniques.

Media applications lend themselves quite naturally to acceleration via SIMD processing, since they contain code sequences with certain unique characteristics–

- Small, native data types (e.g., 8-bit pixels, 16-bit audio samples).
- Recurring operations performed on a large array of data items.
- Compute-intensive.

To speed up computation on such sequences, ISA extensions pack together the smaller basic data elements, and operate on them in parallel. To a first

approximation, therefore, such SIMD processing with *n* data elements packed together would offer an *n*-fold speedup. Because of various bookkeeping overheads involved (for instance in packing/unpacking), however, the actual speedup obtained is not so obvious.

In 1996, Intel announced the so-called MultiMedia Extensions (MMX) as the first major enhancement to the x86 instruction set since the introduction of the 386 a decade earlier. MMX was a set of *integer* SIMD instructions, i.e., instructions which processed multiple, integral data types simultaneously [5]. It was soon apparent, however, that parallelizing just integer computation was not going to be enough. Floating-point hungry 'visual computing' applications were getting increasingly popular. These applications rely on considerable floating-point number crunching for 3D-geometry processing. As such, the MMX extensions are of limited use in boosting performance on 3D graphics applications. They are restricted to the pixel-processing part of the 3D-geometry pipeline, where no floating-point computation is involved.

It was to tackle this shortcoming that the Streaming SIMD Extensions (SSE, also informally known as the Katmai New Instructions, or KNI) were developed. The scope for the SSE expanded beyond just 3D geometry, however. Intel incorporated feedback from the wish lists of various Independent Software Vendors (ISVs) in defining the new extensions, and added cacheability instructions to increase concurrency between execution and memory access. In all, 70 new instructions and a

corresponding new state were added to the IA-32 architecture. Addition of a new state reduced implementation complexity, eased programming-model issues, and allowed the SSE and MMX or x87-FP instructions to be used concurrently. (Note that MMX state is overlapped with x87-FP state; hence, MMX and normal FP instructions cannot be used concurrently. Also, switching between states is costly, and is best avoided.) SSE state presents itself as a set of 8 128-bit registers (XMM0 through XMM7), which are quite apart from the normal x86/x87 architectural state of the processor.

The SSE instructions fall into the following different groups [8] –

- SIMD FP instructions that operate on four single-precision floating-point numbers –

  This group consists of a new data type (four packed FP values), and a battery of instructions designed to operate on such packed data in parallel. For example, there are new instructions to handle comparison and conditional flow, data manipulation, and conversion instructions. An interesting example of the tailoring of the new extensions to 3D geometry processing is the presence of reciprocal (RCP) and reciprocal square root (RSQRT) instructions, which are basic building-block operations in 3D geometry.
- Scalar FP instructions –

Provided to circumvent a cumbersome programming paradigm resulting out of the use of a stack-based register model for x87-FP, and the flat register model for SSE. Scalar FP uses only one-fourth of the usual 4-wide FP data.

- Cacheability instructions including prefetches into different levels of the cache hierarchy –

  Over the years, as memory struggles to keep up with processor speeds, various techniques have been evolved to soften the blow of increasing memory latency on overall performance. The most common of these has been software pipelining, which can be used to optimize cache performance of loops by loading data several iterations before it is needed. This technique usually relies on using a register to hold the data loaded in a prior iteration, and as such, cannot be used with any effectiveness on a register-starved architecture such as the x86. The solution in this case requires hardware support; such hardware support has been incorporated on the Pentium III in the form of cache prefetch instructions.

- Control instructions.

Obviously, graphics-intensive applications present a classic case for potential SIMD-FP optmization. Hence, we consider a number of popular 3D games, which have been enhanced with the new instructions. We also examine the impact of the cacheability instructions on a set of array- and matrix-processing kernels. Finally, we

compare performance of a benchmark (3D WinBench 2000) on a Pentium II system with that on a Pentium III system to study the speedup afforded by the Pentium III. We use the hardware performance counters on the P6 to obtain our results. We have eschewed the use of VTune as the performance-monitoring tool, however, in favor of more accurate *ad hoc* tools we have developed for the purpose of evaluation.

Our methodology is similar to that used in previous studies along similar lines, particularly that pioneered by Bhandarkar and Ding [1] in their classic paper characterizing the performance of a Pentium Pro processor on SPEC and other benchmarks. There have been some other studies of media applications on a PC platform. Bhargava et al [3] evaluated the MMX technology on DSP and other multimedia applications. Talla and John [4] have presented execution characteristics of multimedia applications on a Pentium II processor. To our knowledge, however, there have been no prior studies focusing on the latest offering of SIMD ISA extensions from Intel, namely, the SSE. As such, the findings in this report represent an attempt to evaluate the efficacy of the Streaming SIMD Extensions in speeding up 3D geometry processing in commercial applications.

The overall organization of this report is as follows:
We present a brief description of SSE implementation on the Pentium III in Chapter 2. Chapter 3 describes our methodology, including the kernels and benchmarks used in this work. It also explains the philosophy behind the performance monitoring

tools developed for this project. Results and analyses are presented in Chapter 4; conclusions are stated and future work projected in Chapter 5.

Chapter 2 : Streaming SIMD Extensions Implementation

In this chapter, we give a brief description of the implementation details of the SSE on the Pentium III processor. The Pentium III, based on the P6 microarchitecture, has the first implementation of the Streaming SIMD Extensions. The SSE algorithms are inherently parallel since the same sequence of operations can be applied concurrently to multiple data elements. Although the

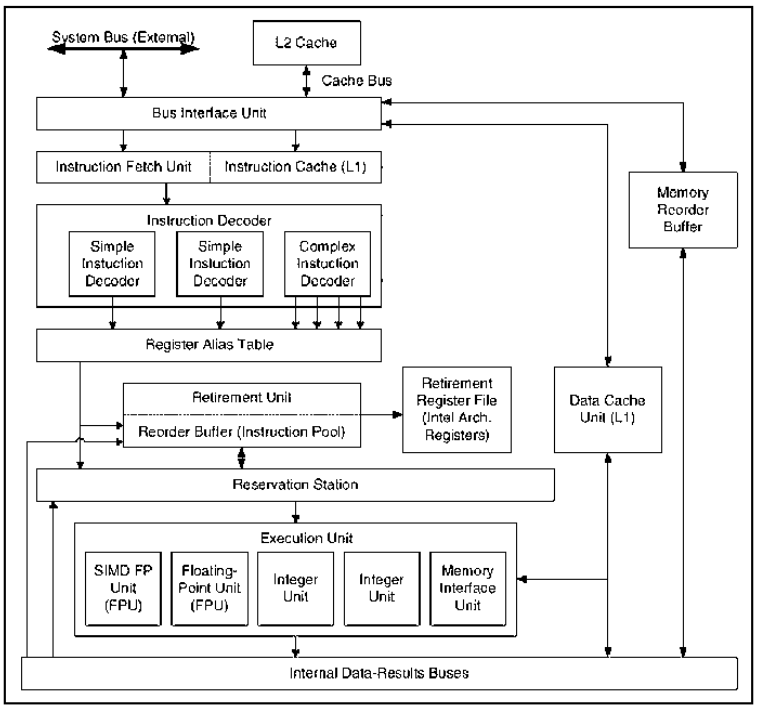


Figure 1: The P6 Microarchitecture

basic P6 out-of-order superscalar microarchitecture is capable of utilizing explicit as well as extracted implicit parallelism, hardware designed specifically to support higher

computation throughput of parallel operations could speed up processing considerably. This is what the PIII adds to the basic P6 core.

The P6 microarchitecture, shown in Figure 1 taken from [9], sports a radical new architecture that is completely different from the Pentium or the 486 before it [11, 12]. The P6 blurs the distinction between CISC and RISC, since its breaks the variable-length x86 instructions into fixed-length RISC-style μ-ops, effectively executing the CISC instructions in a RISC core. This RISC core can then use the tried-and-trusted methods of extracting parallelism from the code such as register renaming and out-of-order execution. This enhancement, coupled with high performance cache and bus designs, allow the P6 to perform well on even large programs. As shown in the figure, the P6 can be divided into an in-order front-end and an out-of-order execution core. Instructions begin execution in-order, execute o-o-o, and retire results in-order [12].

To implement the SSE on the P6, the instruction decoder was modified to translate 4-wide (128-bit) SSE instructions into a pair of 2-wide (64-bit) μ-ops. Since the P6 already has floating point units which process 64 bits at a time (double precision floating point), the extent of modifications required was reduced. Some of the FP units were developed by extending the functionality of existing P6 FP units. Implementing the 128-bit instruction set on the 64-bit datapath limits changes required to the decoder and utilization of existing and new execution units.

Additional features were added to improve the utilization of FP hardware:

- The adder and multiplier were placed on different ports, which means a 64-bit add and multiply can be dispatched at the same time. All new features

have been added on Port 1. Figures 2 and 3, also taken from [9], show the Dispatch/Execute units of the Pentium II and Pentium III respectively. The new units on the Pentium III and the modified P6 units are shown shaded in Figure 3. A new packed adder has been provided on Port 1; the multiplier on Port 0 is a modification of the existing P6 multiplier.

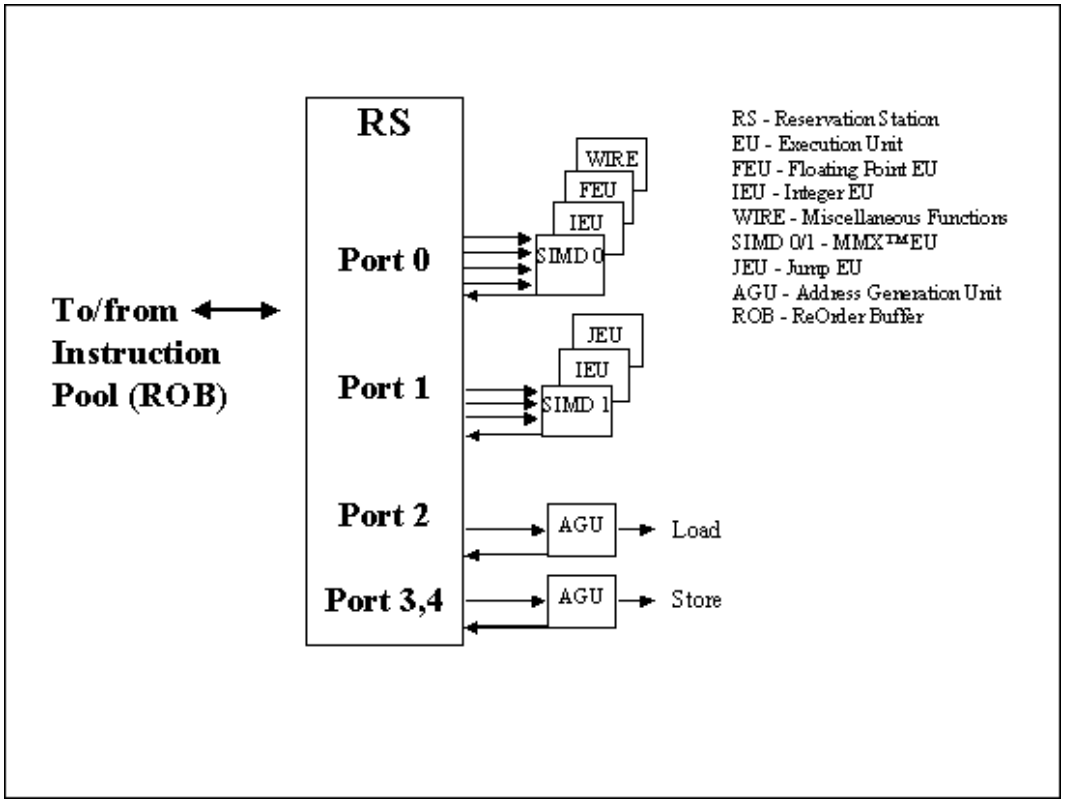


Figure 2: Pentium II Dispatch/Execute Units

- A new shuffle unit has been added to ease and speed up the reorganization/permutation of data within the new SIMD registers.

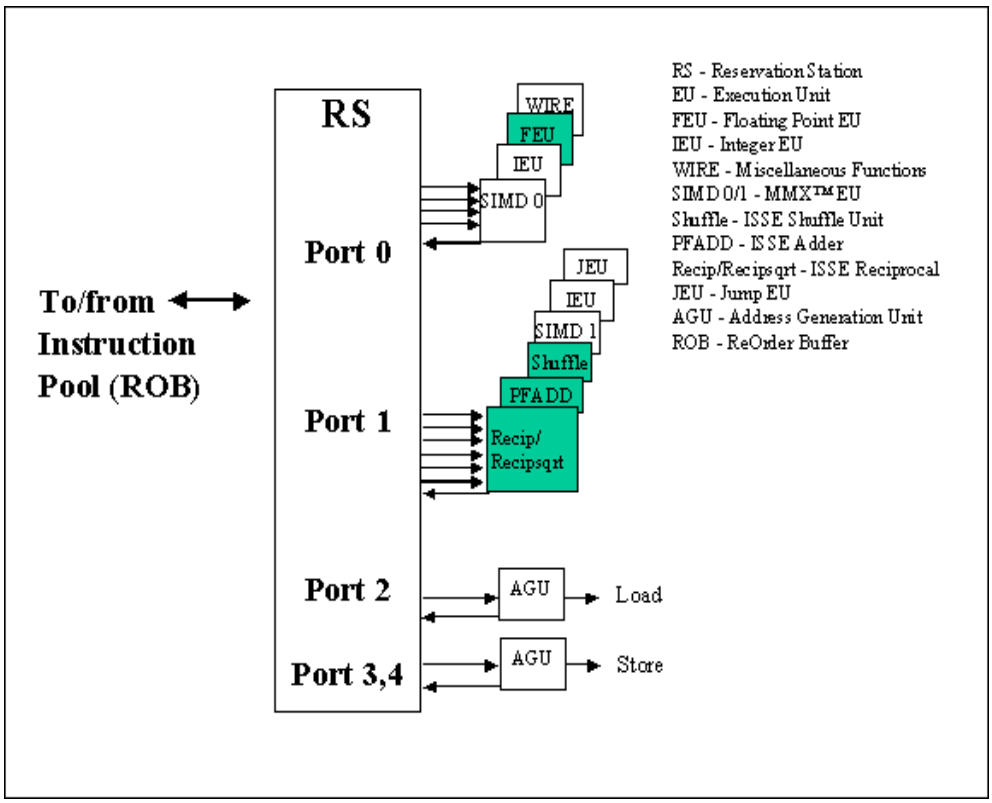


Figure 3: Pentium III Dispatch/Execute units

- In order to improve execution concurrency, the programmer should be provided with a facility to overlap execution of one piece of data with the fetch of the next piece so that the latency of the fetch is hidden by the

  execution time. Furthermore, multimedia workloads are essentially streaming applications in which data are read once, processed, and then discarded. As such, it is desirable to prevent this data from evicting often-needed, long-

term data from the caches. With all this in mind, cacheability instructions were added to the SSE extensions. The prefetch hints were expanded to let the programmer specify the cache hierarchy level where the prefetched data are going to be placed. Complementary instructions were added to perform non-allocating (streaming) stores so that needed data in the cache does not get replaced, and these stores do not generate unnecessary write-allocation. Since prefetches are merely execution hints, they do not generate exceptions or faults. Prefetch implementation is shown in Figure 4. If a load misses in the cache, the instructions following the load are executed, but they cannot retire from the machine till the load returns the data. These instructions may

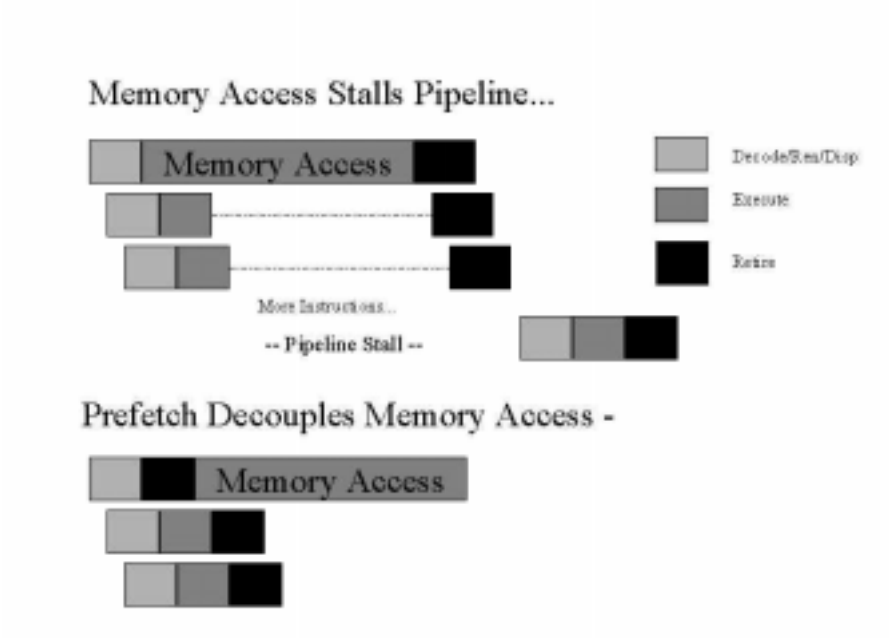


Figure 4: Prefetch Implementation

saturate key resources such as the ROB or the reservation station. To prevent a prefetch load from stalling the pipeline in this way, the prefetch is implemented such that in case of a cache miss, it retires immediately. No retirement and resource saturation stalls are allowed to occur because of the memory access.

For more exhaustive details regarding the implementation and tradeoffs involved in the Pentium III, the interested reader is referred to [8, 9].

Chapter 3 : Methodology, Benchmarks, and Tools

This chapter explains briefly our methodology in this work, the benchmarks that we used, and the tools used to evaluate the extensions. As explained before, this work had three broad thrusts:

- evaluating the cache prefetch instructions,
- characterizing commercial applications which use the SSE extensions, and
- comparing performance of the Pentium II and Pentium III processors.

For evaluating the kernels, we considered the speedup attained by optimizing the kernels with cache prefetch, over the same routines without such optimization. The metric used to evaluate performance was basically the CPI. We focused on small kernels with predictable instruction/branch counts and memory access patterns, so that counter readings could be intuitively justified.

For the case of the other commercial media applications, such clear-cut evaluation was not possible, since non-enhanced versions of the games (which did not use the extensions) were not available. Hence, we have attempted to compare the performance of the enhanced applications with similar non-SSE titles. We have also attempted to correlate the CPI obtained on the applications with the various stalls occurring in the processor, similarly to the results presented in [1]. That paper brings out a very real problem encountered in the study of a superscalar out-of-order architecture: because of all the overlapped execution in the processor, it is not always possible to derive precise cause-effect relationships. We have also noted such discrepancies where they arise; possible causes are suggested in such cases.

Finally, for comparing the performance of the Pentium II and the Pentium III processors, we use the 3D WinBench 2000 suite, a popular PC graphics benchmark.

For all evaluation, we used the performance counters on the P6 core. The specific tools developed are described in section 3.3. Table 13 in the Appendix gives a list of performance metrics that have been measured using the two events to be monitored.

3.1 BENCHMARKS

In this section, we describe the kernels used in the prefetch study, and the games and other applications evaluated for studying SSE-enhancement. We also describe the benchmark suite used in comparing the Pentium II and Pentium III processors.

3.1.1 Kernels

As explained in Chapter 1, software pipelining is not an option for getting execution/memory access concurrency on the x86 architecture. To bring out the efficacy of the prefetch instructions, we specifically chose such kernels as would traditionally be unrolled and software-pipelined in RISC architectures. The kernels that we used are –

1. A simple loop that just increments each element of a linear array –

```
for (i=0; i<SZ; i++)
     for (j=0; j<32; j++)
          a[32*i+j] = a[32*i+j] + 1;
```

2. A loop that adds elements of two arrays together –

```
for (i=0; i<SZ; i++)
     for (j=0; j<32; j++)
          a[32*i+j] = a[32*i+j] + b[32*i+j];
```

3. A matrix multiplication kernel –

```
for (i=0; i<SZ; i++)    {
     for (j=0; j<SZ; j++)    {
          c[i][j] = 0;
          for (k=0; k<SZ; k++)
               c[i][j] += a[i][k] * b[k][j];
     }
}
```

In the first two cases, *a* and *b* are character arrays which are aligned on a cacheline in memory. The inner loop processes the 32 characters in a single cacheline, and the outer loop iterates over cachelines. The third kernel is a simple matrix multiplication kernel, which gives a result matrix from two source matrices. Note that in the first two cases, only one loop is really needed; the nesting has been used only to show the cacheline-by-cacheline processing of the data in exaggerated relief.

3.1.2 Commercial Applications

Along with the cache prefetch studies, we have also analyzed some commercial media applications, which have been enhanced by the SSE. Not

unexpectedly, most of these applications are 3D games, in keeping with the observation that the extensions were designed specifically to speed up 3D geometry processing. Note that a lot of games rely on the DirectX set of APIs for Windows 95/98; which is a family of APIs designed to allow applications to perform certain i/o operations directly, bypassing the OS and avoiding the overhead of a normal system call. Direct3D, the 3D-geometry engine for DirectX, is tuned for the Pentium III processor, speeding up 3D rendering tasks such as scaling, rotation, lighting and other effects. Thus, games that rely on DirectX for rendering automatically get optimized for the Pentium III processor.

The following games/applications were considered:

- **Quake 3 Arena** – The hugely popular first-person shooter game. Processor vendors and graphics accelerator manufacturers often use this benchmark as a standard gaming benchmark.

- **FreeSpace** – A spaceship combat simulation.

- **Expendable** – A 3D arcade game. Both FreeSpace and Expendable have been specifically optimized for the SSE. Thus, they will not run on a Pentium II or lower processors.

- **Metastream plugin** for the Internet Explorer browser – This is a plugin for the Internet Explorer 4 web browser, which allows the user to interactively view 3-dimensional objects on a website [17]. Using the plugin, the user can grab at an object with the mouse, turn it around in any direction, and set it spinning on an arbitrary axis. The plugin is basically intended to be used for

allowing online consumers to interact with a virtual product on an e-commerce site, much as they might in a store. The Metastream plugin provides realistic rendering with physically correct lighting, reflections, and shadows, to provide accurate product representations. We evaluated a typical session with the plugin, where the user interacts with a 3D model of a spaceship.

Besides the four SSE-enhanced applications mentioned above, we also evaluated three non-SSE applications to compare their characteristics with those of the SSE-enhanced apps. These applications were:

- **Doom** – Arguably the first popular first-person shooter game.

- **Quake** – The successor to Doom; this is a very similar game, but is highly optimized for the x86/Windows platform.

- **Windows Media Player** – This is the standard Windows media (audio/video) player that streams media content over the Internet. We evaluated the Media Player as a non-SSE Web-based counterpart to the Metastream plugin described above.

All of the media applications were tested on Windows 98, which is the only Windows platform that supports the latest version (7.0) of DirectX. The only exception was the Metastream plugin, which was tested on Windows NT, for reasons explained later.

3.1.3 A Benchmark for Graphics Subsystem Performance – 3D WinBench

To compare the Pentium II and Pentium III processors, we used the 3D WinBench 2000 benchmark suite [16] created by the ZD Benchmark Operation (ZDBOp), an independent developer of benchmark software. 3D WinBench measures a PC's 3D-system performance under Windows 98 or Windows 2000 using the Direct 3D interface.

We digress briefly to discuss the basics of 3D graphics processing in today's computer systems. Figure 5 shows a typical 3D-pipeline structure and its associated application-level components.

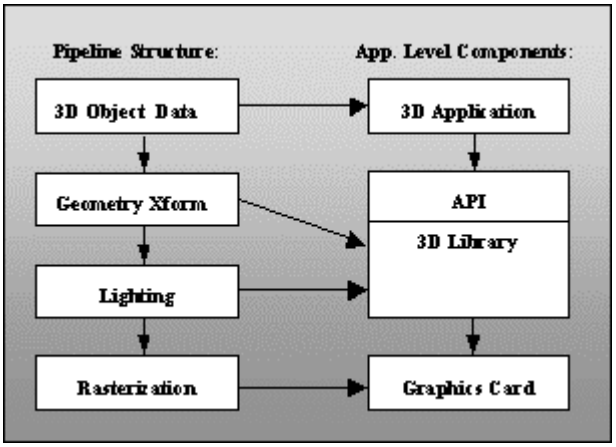


Figure 5: The 3D Graphics Pipeline

- The 3D object data is supplied by the application, according to the object that needs to be drawn on screen. This is usually in the form of vertex data for the triangles that the object is composed of.

- Geometric transformations are carried out on the vertices – there are basically matrix operations that orient the 3D object in space in the desired manner, translating, shearing, and scaling the object as required.
- Lighting calculations are then done to calculate the level (intensity, color etc.) of the illumination on the object. This is done in accordance with the lighting model being used, and the reflectivity of the object. This step involves considerable floating point processing to normalize vectors and compute dot products.
- Finally, the 3D object that has been transformed and appropriately lit, must be rasterized (converted into a 2D pixel representation) on the computer screen.

The first step in the pipeline involves the application. The next two steps (geometry transformations and lighting) are done using the functions provided by a popular 3D library (such as OpenGL or Direct3D), or in some cases, by the application itself. The last step (rasterization) is handled by a 3D graphics accelerator. An increase in graphics accelerator performance means that the 3D libraries developed to deliver the geometry information to the card must also increase in performance to keep the division of work in balance. The Streaming SIMD Extensions were developed, in part, to increase the throughput of the geometry and lighting calculations [10].

Direct3D is the 3D library most used on Windows platforms. Direct3D consists of two different APIs: Retained Mode and Immediate Mode. In Retained Mode, the Direct3D software actually builds frames based on application information. In Immediate Mode, Direct3D gets complete frames from the application and sends them to the graphics adapter. A characteristic of Direct3D

applications is that they must use triangles to display their images; therefore each frame consists of a number of triangles of varying shapes, sizes, and colors.

Retained Mode sits on top of Immediate Mode. There are no Retained Mode-specific features that a display adapter could accelerate. Like most Direct3D games, 3D WinBench 2000 uses the Immediate Mode API of Windows' Direct3D interface.

The 3D WinBench suite contains several tests designed to exercise all of the components of a PC's graphics subsystem –

- The Direct3D API (the benchmark requires DirectX 7.0)
- Monitor
- 3D Accelerator
- 3D Driver
- The processor subsystem
- RAM
- The bus between the graphics adapter and the processor subsystem

We used the two tests in the suite, which provide a quantitative result representing the overall performance of a system – the 3D WinMark 2000 and 3D WinBench 2000 Processor Tests.

The 3D WinMark 2000 suite includes a series of scene tests that vary in both complexity –the number of triangles they use to form their images – and the number of quality-enhancing options (such as fog and specular highlights) they employ. Each test flies through a scene using a predefined path and measures the rendering speed

in frames per second (fps). This suite returns an overall frames-per-second 3D WinMark result summarizing the computer's performance. This is just the arithmetic mean of the fps scores for each of the individual scene tests.

The 3D WinBench Processor Tests perform the same processing as the WinMark tests, except that the rasterization part of the 3D pipeline is omitted. The tests use a 'NULL' graphics device, that is, one that accepts frames generated and does nothing. Therefore, these tests just evaluate the *processor* part of the graphics pipeline, isolating it from the graphics accelerator. Results are given in terms of triangles rendered per second (Δps), which is the harmonic mean of the scores for the individual tests. It is these tests that give the speedup obtained *just* from the processor, and it is these tests that are expected to reveal the raw benefit, if any, obtained from the SSE extensions.

3.2 PLATFORM

The platform used for evaluation was a Dell Precision 410 system with a Pentium III processor at 500 MHz with 16KB L1 instruction and data caches and 512kB L2, running Win98/NT 4.0. The system had 1GB of main memory, and used an NVidia Riva TNT2 graphics accelerator with 32MB memory. This section describes briefly the architecture of the Windows family of operating systems, and then explains the philosophy behind the performance monitoring tools developed for the purpose of evaluation.

3.2.1 The Windows Family of Operating Systems

The Windows family operating systems are pre-emptive, multitasking, virtual memory operating systems. They come in two flavors – Windows 98 and Windows

NT. Win98 and NT appear identical from the perspective of the Win32 API, but differ radically in their internal organization.

Win98 has a Virtual-Machine architecture, where the system is organized as a set of Virtual Machines managed by a Virtual Machine Manager (VMM). This structure is necessitated by the requirement that Win98 has to be able to run *all* of the legacy DOS applications, which assume that they own the machine and are prone to directly access i/o ports and memory.

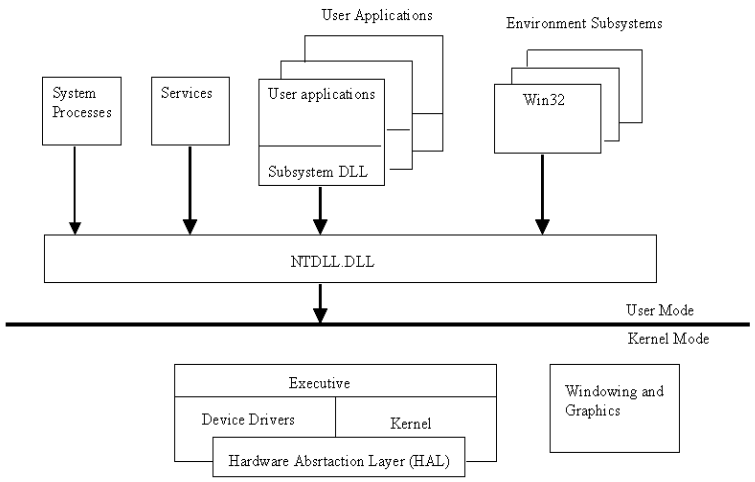


Figure 6: Windows NT Architecture

Windows NT has a more traditional structure that merges attributes of a layered OS with a microkernel OS [12]. Performance-sensitive operating system components run in the privileged processor mode (called *kernel* mode), and user applications run in the less privileged, or *user* mode. A simplified view of the NT architecture is shown in Figure 6. The components above the bold line in the figure

are user mode processes, and the components below the line are OS services. Architecture specific functions such as thread scheduling and dispatch are implemented in the NT kernel. More details of NT internals can be found elsewhere [12].

Device drivers are loadable kernel mode modules that interface between the I/O system and the relevant hardware. Because of the vastly different internal structures of the two systems, the native device driver interfaces for the two systems are also different. Win98 employs the so-called V*irtual Device Drivers* or VxDs, which call upon the services of the Virtual Machine Manager to perform their tasks. WinNT, on the other hand, uses *Kernel Mode* drivers, that call upon the kernel and HAL (Hardware Abstraction Layer) services to interface with hardware. In order to unify the programming model for drivers in the two systems, Win98 also supports the Windows Driver Model (WDM) architecture, which is a unifying programming paradigm designed to ease the driver development process and protability issues across the two sets of operating systems. WDM essentially presents to Win98 drivers a kernel environment almost identical to that in NT. As such, it is possible, with some reservations, to develop a driver for NT and recompile it to function under Win98. This is the approach that we have taken. The following paragraphs explain the driver model under Windows NT with the understanding that everything applies without change to Win98 except where noted.

As mentioned earlier, device drivers are kernel mode modules that provide certain services to user-level applications. Device drivers become a part of the NT kernel and run in kernel mode; therefore, they can execute any of the processor's privileged instructions. A kernel-mode device driver is the only means to add to the operating system, user-written code, which needs to execute privileged instructions.

A kernel mode driver was therefore used in this work to execute the privileged WRMSR and RDMSR instructions that write to, and read from the performance monitoring Model Specific Registers (MSRs), respectively. The basic structure of the driver is described in the next section; a brief discussion of NT thread scheduling follows next.

Processes, Threads and Scheduling

A *process* in Windows NT comprises an executable program that contains the initial code and data. A process has its own virtual address space, and is identified by a unique number called the *Process ID*. A *thread* is the entity within a process that NT schedules for execution. A thread includes the contents of a volatile set of registers that represent the thread's *context* of execution. Every process has at least one thread of execution.

Windows NT implements a pre-emptive, priority driven thread-scheduling system– the highest priority ready thread always runs. Note that NT schedules at the thread level and not at the process level. When a thread is dispatched to run, it runs for a length of time called a *quantum*. A quantum is the length of time that a thread is allowed to run undisturbed before NT interrupts (or *preempts*) the thread to check if any other threads at a higher priority level than this thread are ready to run. If there are any, then NT initiates a context switch. A thread quantum on Windows NT WorkStation is around 20 ms.

Thread dispatch can be initiated by any of the following events:

- A new thread becomes ready to execute.
- The current thread leaves the running state because its time quantum ends.

- A thread's priority changes, either because of a system service call or because NT itself changes the priority value.

This preemptive thread scheduling imposes some bearings on a performance-monitoring instrument based on the performance counters. If the counters are used in a "start-stop" mode, i.e., starting them when the application begins execution and stopping them when the application stops, then there is a good chance that the counts will be corrupted by any context switches occurring during the time that the application runs. It would therefore be desirable to have a measuring instrument that has a "stop-and-go" rather than a "start-stop" characteristic – that is, the counters should ideally count only for the duration that the application under test runs. Past studies that have relied on a start-stop method have been plagued by corruption of counts in the multitasking environment. This difficulty compounds the problem mentioned earlier of not being able to pinpoint causes of CPI degradation.

As explained in the next section, it has been possible to develop a fairly accurate counter tool for Windows NT. WDM limitations on Win98 made it impossible to port the tool over directly to that operating system; hence we had to fall back on a more simplistic "start-stop" version. We have tried to alleviate the distortion of counts as far as possible in this case, by ensuring that the process under test is the only process that is running while measurements are being made.

3.3 TOOLS

This section describes the tools developed, that is, the performance monitoring device drivers for Win98 and NT. Another issue that arose during the benchmarking of games was that of ensuring *repeatability* of user actions during

successive runs of the game. A few of the games do provide a 'demo' feature that keeps looping through a demo at startup. However, others don't, and to ensure comparable counts on successive runs, it was felt that it would be necessary to provide exactly the same user input every time. With this intent, we developed a tool that would record user input (mouse movements/clicks, keystrokes) on a sample run of an application, and play it back later on demand. This tool uses the *journaling* facility provided by the Windows API, explained later in this section. Unfortunately, it turned out that the tool could not be used on the non-demo games at all – playback was erratic and kept skipping over arbitrary items of input. We believe this is because games use DirectX for input, which probably bypasses the standard Windows input-queue mechanism. Nevertheless, we were able to use the tool on the Metastream plugin – since that plugs in directly into the browser, which does use the normal input queue.

3.3.1 PMON: The Performance Monitoring Device Driver

Traditionally, VTune, Intel's visual performance tuning tool, has been used to evaluate any application on the P6 microarchitecture. VTune, however, is cumbersome and bulky, and as a performance-monitoring tool, it is not very useful – it violates the cardinal rule that a performance measuring instrument should be as *non-invasive* as possible so as not to perturb the system being evaluated. It was necessary, therefore, to create a "lightweight" performance-monitoring instrument that would not interfere with the application being monitored.

Since the performance counters can be programmed/read only via privileged instructions, it was necessary to develop a kernel mode device driver, which would execute those instructions. This section describes the philosophy behind the design

and the basic structure of the performance monitoring device driver. As explained earlier, this section focuses on Windows NT, since the driver for Win98 is merely a stripped-down version of the NT driver.

3.3.1.1 Design Philosophy

A primary objective behind the driver design was to have it stop the counters when the process under test was context-switched out, and start them again when the process was switched back in. This was accomplished using the *timer objects* provided by the NT kernel, as discussed shortly.

The system timer tick interrupt in NT occurs roughly every 10 ms. It is the clock handler that decides if the currently running thread has run long enough, and if so, whether it should be pre-empted to let another thread run. It is then logical to imagine that if the driver too could be awakened on every timer tick, it could check if the process under observation was running or not, and either start or stop the counters depending on the current scheduling status for the process.

It can be seen from the following description of the driver structure how this functionality has been woven into the device driver.

3.3.1.2 Driver Structure

A Windows NT Kernel Mode/WDM Device Driver has the following main components [14] –

- An **initialization routine,** which is executed when the driver is first loaded.

- A set of **dispatch routines,** which are the main functions that a driver provides. It is via device I/O control calls in a dispatch routine that the driver communicates with the host Win32 application that issues those calls.

- An **interrupt service routine (ISR)** for drivers that service external device interrupts.

- An interrupt-servicing **DPC routine** that is used in conjunction with an ISR.

Since this driver was not connected with any hardware device, the last two routines are not relevant.

The basic operation of the driver is then as follows:

- The host application, which actually uses the driver services, spawns the process to be monitored and passes the driver the process ID of the newly created process. The driver saves this ID in an internal data structure.

- The driver also resets the counters to zero and configures them to count desired events.

- To allow itself to be waken up every timer tick, the driver configures a kernel timer object to time out on every timer tick, and establishes a routine to be

called when the timer fires. It is this timer routine that implements the "selective-count" functionality of the driver.

- On every timer tick (i.e., every 10 ms or so) the timer routine extracts the process ID of the currently running thread from an internal system data structure, and compares it with the process ID of the process being measured. (This was passed to the driver by the host application earlier.) The driver also maintains a memory of the last comparison, i.e., the process ID match status the last time that the timer fired. Based on these two results, the driver takes the following steps:

Did Process IDs match on last timer tick?	**Do Process IDs match on current timer tick?**	**Update value for LastProcessIDMatch flag**	**Action**
N	N	N	–
N	Y	Y	Re-enable counters
Y	N	N	Disable counters
Y	Y	Y	–

By disabling the counters when the process is not running, the driver can maintain an accurate, selective count of events for the process under observation.

This feature is absent in the Win98 version of the driver because of WDM limitations on the Win98 platform. In particular, there appears to be no way of getting at the Process ID in kernel mode using WDM. As such, this scheme becomes unimplementable, since getting the Process ID inside the timer routine is crucial. We therefore had to restrict ourselves to a less sophisticated "start-stop" version of the driver on Win98.

3.3.2 The Journaling Tool

In evaluating the games, a key issue that arose was that of ensuring repeatability of measurements. Some of the games did provide demos that kept looping at startup, which we could use for taking counter readings. However, some of them did not, and we wanted a way to send the same user input over and over to the game. An approach that did seem plausible was that of getting some kind of a "keyboard script" to play back a series of keystrokes recorded on a sample run of the game. This would help ensure nearly identical sequences of execution on repeated runs of the game, therefore yielding comparable counts. (Note that multiple runs are needed to record data involving all events, since only two events can be recorded by the two counters during every run.)

The Windows Win32 API offers a mechanism (Win32 Hooks) to intercept events (mouse actions, keystrokes) before they reach an application. A function attached to the keyboard hook, for instance, sees all keyboard input before it passes on to an application. This function can act on events, and modify or discard them. (The function is therefore called a "filter" function.) For Windows to call a filter function the function must be installed; the API provides functions to set and unhook a filter function.

Windows provides a variety of hooks, among them being hooks to process, modify or remove keyboard and mouse events. There is also a pair of hooks that is tailor-made for the purpose we had in mind: the "journal" hooks, which allow for recording and playback of a series of keyboard and mouse events. (The recording and playback, in Windows parlance, are collectively referred to as "journaling".)

Journaling hooks basically allow a list of messages to be built while recording. Every time an input event occurs, either on a mouse move or a keyboard hit, Windows calls the filter function, passing it a message structure identifying the event that caused Windows to invoke the function. A timestamp is included in every such message structure. The time delta between successive messages can be given to Windows during playback, so that the record speed is preserved while playback.

We developed a simple journal recording and playback application that would record all events on a sample run of an application and store them in a file; these events would be loaded and played back on demand. While sound in principle, this technique did not really work in practice. The tool gave very erratic behavior on games, and accurate playback/recording could not be achieved. We believe that to be due to the fact that games use DirectX for input. This presumably bypasses the normal Windows input mechanisms, leading to faulty behavior of the journaling tool.

The journaling approach, therefore, had to be abandoned in the case of games. We had to resort to measuring counts by just manually playing the game repeatedly, in spite of the fact that this must reduce the correlation between counts obtained on different runs of the program. Interestingly enough, there is not much variation in different runs in counts of events such as instructions retired. That is because, with a CPU running at 500 MHz, the number of instructions executed in 60 seconds of a run of a game is well into the billions. Slightly different playing styles do not cause a substantial difference in the number of instructions counted. However, we were able to use the journaling technique in evaluating the Metastream browser plugin, since that is a normal Windows application. We recorded a series of mouse movements where we grabbed a 3-dimensional object with the mouse, turned it

around, set it spinning, and so on. We could play this series of events over and over to get the event counts we needed.

In passing, we note one important point regarding the use of journaling – while using journaling, it is essential that the more accurate selective-count version of the driver be used. In the journaling process, the parent application spawns the app to be monitored. It then loads the recorded events into memory and installs the journal playback hook, which then begins playback of the recorded events. Because of the specific mechanics of journaling, however, the parent application cannot go to sleep till the spawned app ends as in the usual case. It has to keep a *message loop* running till playback completes. This message loop has the form

```
while (!*fPlaybackComplete)  {
      if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))  {
            GetMessage(&msg, NULL, 0, 0);
            DispatchMessage(&msg);
      }
}
```

and its job is to queue any input that the user might make while the playback is going on.

As such, this will drastically affect the counts measured on the application to be monitored. We have therefore run the MetaStream tests only on Windows NT, for which the more specialized, selective version of the performance-monitoring driver is available.

Chapter 4 : Results and Analysis

This chapter presents detailed results and analysis of the observations made during the course of this study. The first section presents the results of the prefetch characterization on kernels, the second section gives the results on the applications, and the third shows the results of comparing the Pentium II with the Pentium III.

4.1 PREFETCH OPTIMIZATION

We examined the effect of prefetch instructions on the three kernels described earlier. In general, the basic metric for speedup has been the CPI. Contrary to initial expectations, the number of cache misses could not be used to demonstrate difference in performance, since it turns out that the prefetch instructions themselves miss the caches; therefore, the cache miss counts in the two cases are about the same. As such, only the number of clocks could indicate any performance improvement, if any.

While conducting the measurements, the arrays are initially initialized with random numbers. This act of initialization itself can cache the entire arrays for small array sizes; to avoid this from corrupting the measurements, the caches were invalidated using the WBINVD instruction before every run. Note that two kinds of prefetch instructions are implemented on the PIII, one, which prefetches to the L1 only, and another that prefetches to the L1 as well as L2. In this study, the prefetcht0 instruction was used, which prefetches to both caches.

Measurements were taken with both – Debug (no compiler optimization) and Release (code optimized for speed) builds with the Intel C/C++ compiler. This

compiler also provides an option that is supposed to insert prefetches automatically (-Qpf); however, this option was not found to produce any effect.

4.1.1 Simple Loop with Single Array

In this case, a single prefetch for the next cacheline is inserted in the outer loop. Thus, while the 32 bytes in the current line are being processed, the external bus services a miss on the next cacheline, which would ordinarily have resulted when the first load from a byte in that next line was issued. As such, the miss is serviced in parallel with computation, resulting in the memory latency effectively being hidden by the prefetch. This is shown in the code segment below:

```
for (i=0; i<SZ; i++)  {
      _mm_prefetch(&a[32*(i+1)]);  // pref nxt a block
      for (j=0; j<32; j++)
            a[32*i+j] = a[32*i+j] + 1;
}
```

The results are tabulated in Table 1.

Metric	**Debug Build** (No Compiler optimizations)		**Release Build** (Optimized for speed)	
	Prefetch	**Prefetcht0**	**No prefetch**	**Prefetcht0**
# of instructions	4,885,224	4,923,387	1,651,058	1,721,059
# of clocks	5,417,483	4,686,505	2,378,334	1,812,247
CPI	1.109	0.951	1.44	1.052

Table 1: Prefetch Performance – Single Array

As can be seen from the table, inserting a prefetch in a simple single array loop improves performance considerably. Performance improvement in terms of number of clocks improves by 13.5% without and almost 25% with optimizations on. IPC improvement is further shown in Figure 7.

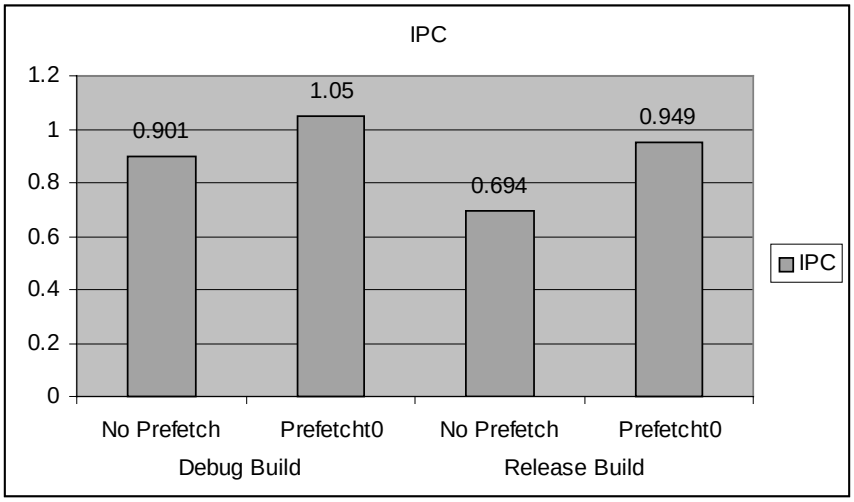


Figure 7: IPC Improvement for Single Array

4.1.2 Simple Loop with Two Arrays

This is similar to the first case, but with processing involving adding elements of one array to the corresponding elements of another array. Since two arrays are involved, the kernel becomes predominantly memory-bound rather than CPU-bound. As such, it is to be expected that prefetch would be even more effective in hiding memory latency and improving performance than in the previous case. This is borne out from the numbers obtained. The code segment for this case is shown below:

```
for (i=0; i<SZ; i++)    {
      _mm_prefetch(&a[32*(i+1)]);  // pref nxt a block
      _mm_prefetch(&b[32*(i+1)]);  // pref nxt b block
      for (j=0; j<32; j++)
            a[32*i+j] = a[32*i+j] + b[32*i+j];
}
```

Results are tabulated in Table 2.

Metric	**Debug Build** (No Compiler optimizations)		**Release Build** (Optimized for speed)	
	No prefetch	**Prefetcht0**	**No prefetch**	**Prefetcht0**
# of instructions	6,485,568	6,565,500	2,311,669	2,411,058
# of clocks	6,697,175	5,779,257	2,742,924	1,928,534
CPI	1.032	0.88	1.186	0.80

Table 2: Prefetch Performance – Two Arrays

Performance now increases by over 15% without optimizations, and almost 30% with optimizations on. The IPC improvement can be seen graphically in Figure 8.

It can be seen that as an application becomes more memory-bound, prefetch can help performance greatly. Performance figures obtained above are very good because these example loops have been quite contrived; even so, performance on real applications can be expected to have a positive boost because of prefetching.

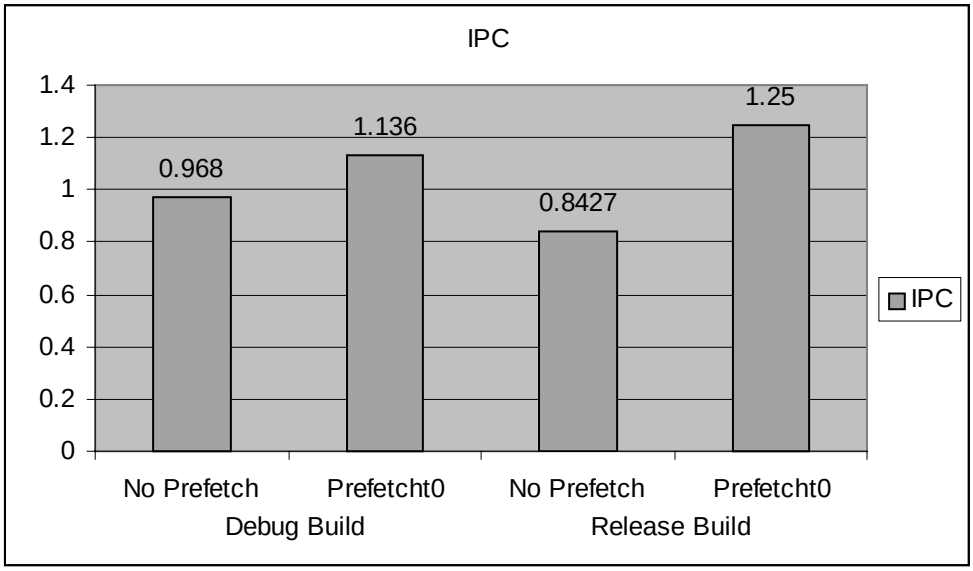


Figure 8: IPC Improvement for Two Arrays

4.1.3 Matrix Multiplication

Matrix multiplication has been a rather tricky kernel to optimize for cache prefetch. For one thing, the data access patterns are much more irregular compared to the rather straightforward patterns in the preceding two cases. Secondly, unlike the first two cases, matrix multiplication exhibits *temporal* locality in addition to the normal spatial locality of the preceding two loops. Thus, data, once cached, is *reused* as rows and columns are multiplied together. As a result, the adverse effect of a cache miss does not impact performance so much, since the cost of that miss is amortized over the number of times that that line is reused for computation. Predictably, therefore, the improvement in this case has not been as much as in the previous cases.

To introduce cache prefetches effectively, we need to consider the misses caused in a matrix multiplication. In order to do so, consider the case shown in Figure 9:

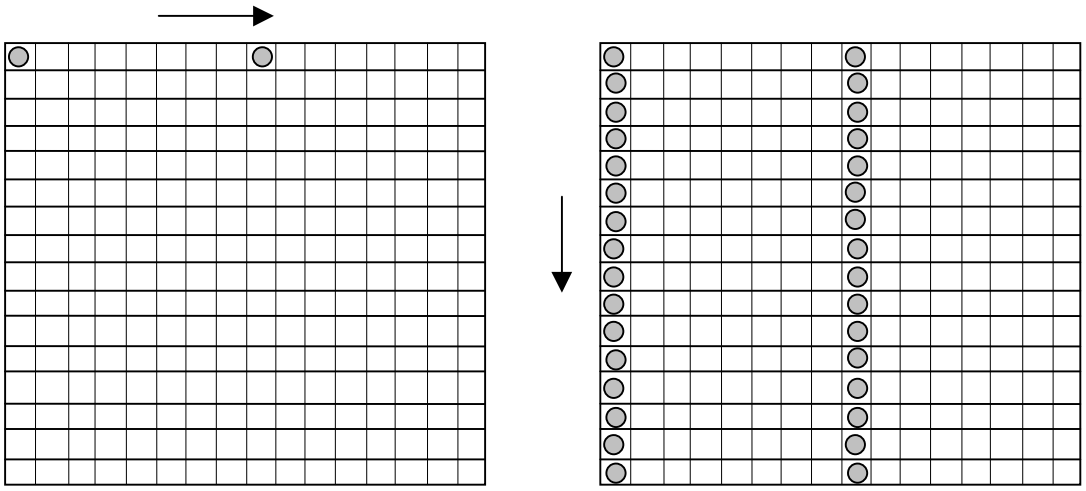


Figure 9: Cache Behavior in Matrix Multiplication

a and *b* are two integer matrices of size 16 × 16. In computing the first row of the result matrix *c*, we traverse the first row of *a* and all columns of *b*, doing a point-by-point dot product of elements in row 1 of *a* and each column of *b*. The cache misses incurred in doing so are indicated by the filled dots in the matrices. Since each integer is 4 bytes, eight integers fit in a single cacheline. As a result, traversing the first row of *a* incurs the two misses shown, and going down the columns of *b* incurs 16 misses for every 8th column. In addition to the misses shown above, we incur additional misses in writing the result matrix *c*, since the P6 uses a write back policy with write allocate on writes.

In optimizing for prefetch, therefore, we would have to consider the effect of all three misses. In order to insert prefetches efficiently, we needed to unroll some of the inner loops. We noticed that just the effect of this unrolling improved performance by reducing the total number of instructions generated by the compiler as well as the clocks. Inserting the prefetches after this improved performance additionally to a certain extent, though, as noted earlier, the speedup gained was not as prominent as that noticed in the earlier cases.

We tried a number of different prefetch strategies, first trying to prefetch just for matrix *a* in the figure; then adding a prefetch that would save the write-allocate of *c*. In both of these cases, however, there was no real improvement in performance. This follows from the fact that there are just two misses for each of *a* and *c* in computing a single row of the result *c* in the 16×16 case. As such, we cannot expect there to be much improvement by prefetching.

Some noticeable improvement manifests only when we insert prefetches for the matrix *b*, to eliminate the string of misses shown in the figure above. Even then, it can be seen that these misses will be incurred only while the calculation of the first row of matrix *c*, since thereafter (at least for small matrix sizes) the entire matrix *b* comes into the cache. Two strategies were tried for prefetching matrix *b*–

- Prefetching for row n+3 while using the value in row n when going down every 8^{th} column as shown in the figure. This is the method called Pref-I.
- A disadvantage of the above method is that it causes too many prefetches to be inserted in rapid succession possibly leading to load buffer clogging. It is possible to avoid this by *spreading out* the prefetches for every 8^{th} column while processing a few columns prior to the eighth. In the present case, for a

matrix of size SZ, we inserted prefetches for SZ/4 elements of the 8th column while processing each of the 4 columns before the 8th column. This is the method named Pref-II.

The results in these two cases for Debug builds are shown in Tables 3 and 4, for matrices of size 64 and 128.

Metric (Matrix size 64)	**Unrolled Inner Loop (no prefetch)**	**Pref-I**	**Pref-II**
# of instructions	6,515,523	6,719,510	6,432,070
# of clocks	7,949,317	8,122,850	7,647,955
CPI	1.22	1.208	1.189

Table 3: Matrix Multiplication (Size 64)

Metric (Matrix size 128)	**Unrolled Inner Loop (no prefetch)**	**Pref-I**	**Pref-II**
# of instructions	51,924,268	53,539,01	51,245,185
# of clocks	87,532,983	82,334,99	83,752,534
CPI	1.68	1.537	1.634

Table 4: Matrix Multiplication (Size 128)

It is seen that though both prefetch methods are better than the code without any prefetches, the expected superior behavior of Pref-II over Pref-I holds only in the case of the matrix of size 64. For larger matrix sizes, the matrices no longer fit in the L1. The deterioration of performance for the larger matrix is probably because data prefetched much in advance (say 4 columns before it was needed) might be getting evicted from conflict misses with some data brought in before the prefetched data is actually used. This could result in some of the data being fetched *twice* into the processor's cache, once by the prefetch and the next time when it is brought in by the actual load.

4.1.4 Prefetch vs. Infinite Cache

Finally, we examine how prefetch compares with the ideal case of an infinite cache, where all data accesses result in cache hits. We approximate the ideal cache case by ensuring that all data fits in the L1; thereby minimizing deviations from the ideal. The worst-case scenario of no cache is achieved by invalidating all caches using the WBINVD instruction. This is compared with the infinite cache version wherein the entire array is prefetched into the cache without cache invalidation before each iteration. The prefetch case is tested by invalidating the caches and then prefetching normally.

4.1.4.1 Simple Loop with Single Array

In this case, the array size is kept at 4kb, so that it fits in a single way of the 16kb 4-way set-associative L1. This allows maximum leeway to accommodate conflict mappings, should they arise because of context switches during execution of the kernel. The readings obtained are tabulated in Table 5:

Metric	**Debug Build** (No Compiler optimizations)			**Release Build** (Optimized for speed)		
	Worst Case	**Ideal Case**	**Prefetch**	**Worst Case**	**Ideal Case**	**Prefetch**
Instrns	63526	63526	64039	22178	22434	23075
Clocks	70658	63084	65632	34721	26541	27449
CPI	1.11	0.99	1.025	1.565	1.183	1.189

Table 5: Max Benefit – Single Array

It can be seen that prefetch has much better performance than the worst case, and closely approaches the ideal case performance. (Degradation is only 4% in the debug build and 3% with the release build.)

4.1.4.2 Simple Loop with Two Arrays

In keeping with the reasoning suggested in the above case, the size of each array in this case is kept at 2kb. The same set of readings is taken; the results are tabulated in Table 6. Note that in the case of the release build, prefetch performs even worse than the "worst" case. This is probably because the data-set size is too small and the optimized code structure generated by the compiler is such that the cost in terms of clocks of the additional prefetch instructions cannot be amortized over the time it takes for processing the entire array. This discrepancy was seen to vanish if the data set size was increased to 4kb for each array – in this case, prefetch did give the expected improvement.

Metric	**Debug Build** (No Compiler optimizations)			**Release Build** (Optimized for speed)		
	Worst Case	**Ideal Case**	**Prefetch**	**Worst Case**	**Ideal Case**	**Prefetch**
Instrns	42534	42534	43047	16482	15779	16482
Clocks	46276	40319	41696	16288	15239	16433
CPI	1.087	0.948	0.97	0.988	0.965	0.997

Table 6: Max Benefit – Two Arrays

4.1.4.3 Matrix Multiplication

The results obtained for matrix multiplication on the debug build are shown in Table 7.

Metric	Prefetch Type			
	Worst Case	**Ideal Case**	**Pref-I**	**Pref-II**
Instrns	820263	820263	845991	810151
Clocks	955740	928788	950954	912684
CPI	1.165	1.132	1.124	1.126

Table 7: Max Benefit – Matrix Multiplication

Results for both prefetch schemes mentioned earlier are shown. As expected, the second prefetch scheme performs better than the first and even performs better than the "ideal" case. This latter observation may be reasoned out by noting that matrix multiplication requires a fair amount of memory. It is not possible to pin down all of the three matrices involved in the cache for the duration of the entire matrix multiplication even if they are prefetched initially. As such, data prefetched initially might get evicted from the cache due to conflicts during the execution of the program. When prefetching is used, however, at any time, the data which will be needed next is prefetched regardless of what has happened to data that was used in the past/will be needed in future.

4.2 APPLICATIONS

This section presents detailed results of characterizing the Pentium III P6 core on a number of applications, viz., games and some web applications. In addition to characterizing the SSE-enhanced applications, we have attempted to correlate the CPIs obtained on the various benchmarks with the cycles lost in various stalls on the P6 core.

4.2.1 Characterization of SSE-enhanced Applications

The four SSE-enhanced applications considered were three games, namely, FreeSpace, Expendable, and Quake3, and the MetaStream Internet Explorer plugin. The Pentium III allows counting of the SSE-related events indicated in Table 8.

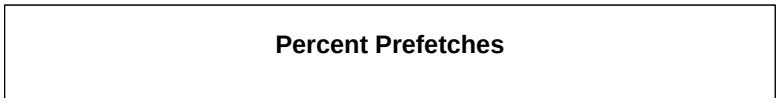
Event	Description
EMON_SSE_INST_RETIRED	Number of SSE instructions retired
EMON_SSE_COMP_INST_RET	Number of SSE computation instructions retired
EMON_SSE_PRE_DISPATCHED	Number of prefetch instructions dispatched
EMON_SSE_PRE_MISS	Number of prefetch instructions that miss all caches

Table 8: SSE Events

Counters can be specified to count either the packed or scalar operations in the first two cases, and the type of prefetch instruction to be used in the other two cases. Even so, it can be seen that the range of events devoted to analyze the SSE instructions is quite limited, especially as compared to the set of events devoted to the MMX technology instructions.

Data obtained for these events for each of the benchmarks is presented here. All games were run on Win98; the Metastream plugin was tested on WinNT, as explained earlier. Figure 10 shows the number of prefetch instructions dispatched, both as a percentage of total instructions as well as memory references. It can be seen that prefetches comprise almost a negligible percentage of the total number of instructions; this figure is only marginally higher with respect to memory references. In particular, Explorer uses no prefetch instructions at all. The number of prefetches used by FreeSpace is too small to be perceptible on the plot. Also, it was seen that all applications use *only* the prefetcht0 form of the prefetch instruction, which prefetches data into all levels of the cache. Neither of the other two types of prefetches is used; nor are streaming store instructions used in any of the applications.

Figure 10: Number of Prefetch Instructions



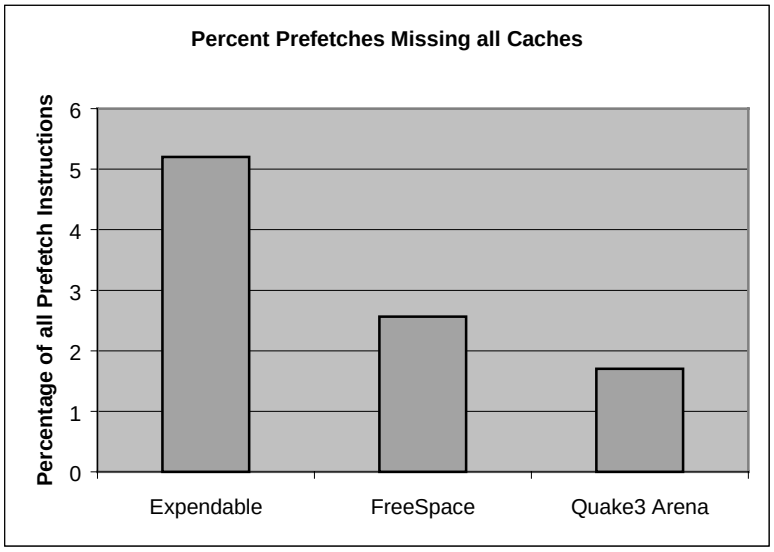


Figure 11: Percent Prefetches Missing All Caches

Figure 11 shows the percentage of prefetch instructions missing all caches. It can be seen that only a small percentage of the prefetches miss all caches. This means that a lot of the time, the data intended to be prefetched into the caches before use is already present in the caches. Evidently, the overlap of the memory-execution pipeline is not as effective as it might be. In particular, Quake3 shows the maximum percentage of prefetches among the benchmarks, yet, less than 2% of these miss in the caches. In this case, therefore, the prefetch instructions are simply consuming resources without providing any significant benefit.

Figure 12 shows the percentage of SSE instructions occurring in the applications. In general, the percentage of SSE instructions is quite small; under 3% in the games. It is maximum for the Internet Explorer plugin, where it approaches 6% of the total number of instructions.

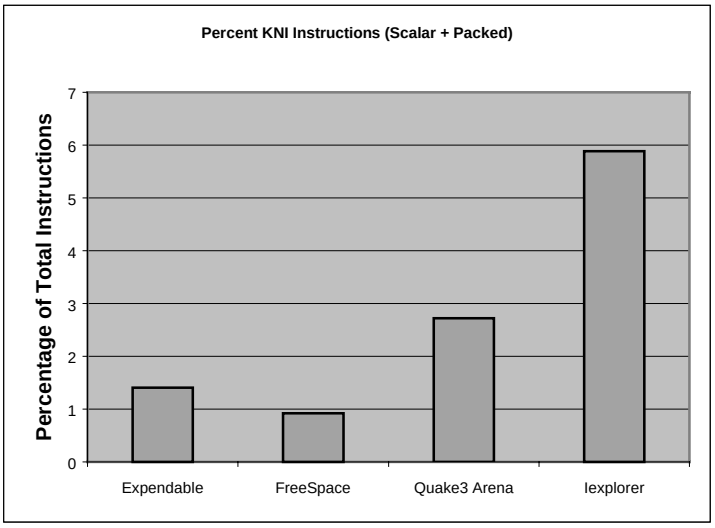


Figure 12: Percent SSE Instructions

The SSE instructions fall into two broad groups, packed and scalar. As explained earlier, the scalar instructions are provided merely to circumvent the cumbersome paradigm involved in using the stack-based x87 architecture along with the flat SSE architecture. It is to be expected that a majority of the SSE instructions would be packed, since SIMD processing is the primary *raison d' être* for SSE. Scalar FP, which has been provided merely for ease of programming, would not be used often. Figure 13 shows that such is indeed the case. In all of the games, most of the SSE instructions are indeed the packed versions; the Metastream plugin takes this to the extreme, where *all* of the instructions are packed. It can be seen that it is only in Quake 3 that the numbers for scalar and packed instructions are comparable. Though Quake 3 has the highest percentage of SSE instructions among the games, the poor prefetch performance and the high percentage of scalar FP instructions would imply that the CPI for Quake 3 will not be markedly improved over that for the other games. CPI variation across the benchmark set is discussed shortly.

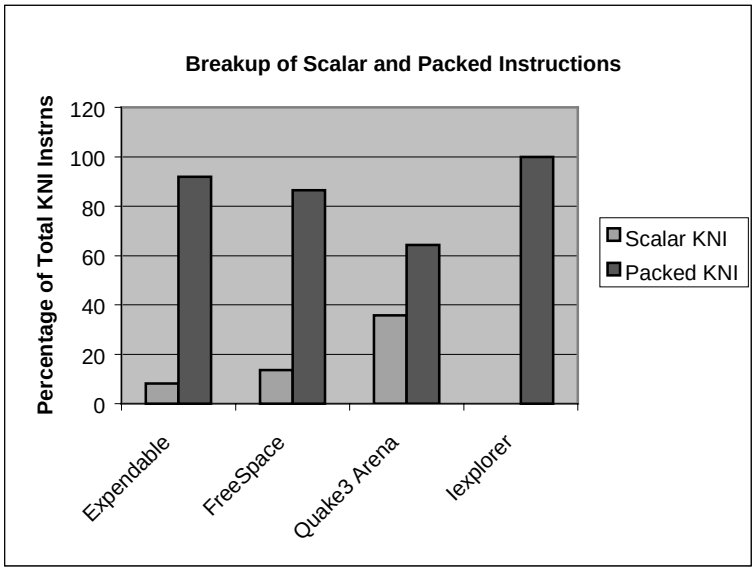


Figure 13: Breakup of SSE Packed and Scalar Instructions

Lastly, we consider the distribution of SSE *computation* instructions (as against the data movement/packing/unpacking instructions) over the set of applications. This plot is shown in Figure 14.

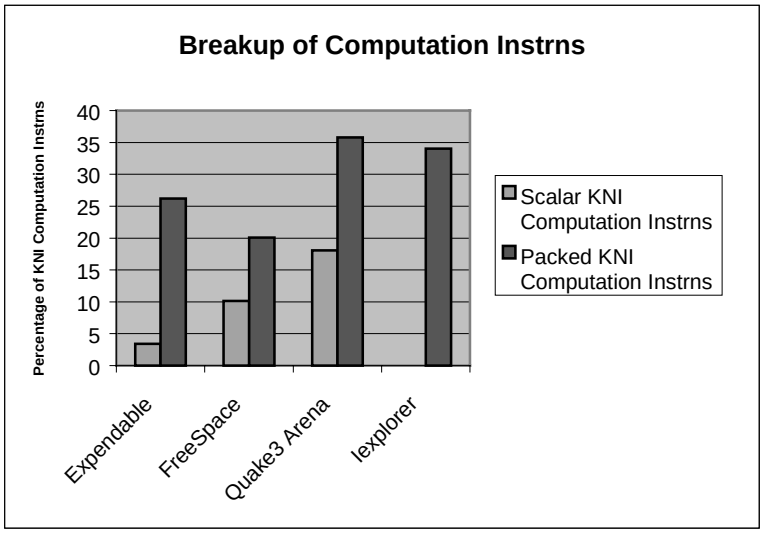


Figure 14: Percentage of SSE Computation Instructions

At most half of the total number of SSE instructions are computation instructions (in Quake 3). More typically, we can see that only about 35-40% of the SSE instructions involve computation. Obviously, the other 60-65% must comprise instructions for packing/unpacking, movement of the packed SIMD data to and from memory, and conversions from one packed format to another (integer to/from FP/scalar/packed, etc.) which is basically the overhead involved in SIMD computation. Thus, we can see that a sizeable fraction of the instructions are being used in loading and formatting the SIMD data for further processing.

4.2.2 Detailed Characterization of Benchmarks on the P6 core

This section presents detailed results of the characterization of the set of benchmarks on the Pentium III. Along with the four SSE-enhanced applications, we also consider three non-enhanced applications for comparing with their enhanced counterparts. We look at the various components of the stalls occurring during the execution of the benchmarks, and attempt to correlate the cycles lost due to stalls with the CPI, as done by Bhandarkar et al [1] in their seminal study.

4.2.2.1 Cycles Per Instruction

Figure 15 shows the CPI obtained on the benchmark suite described in Chapter 3. CPIs for games are generally in the range 1.5 to 1.8. Doom and Internet Explorer have an unusually high CPI for reasons noted later in this section. Windows Media Player shows the least CPI in this benchmark set. This could be

because it basically is just a viewer, which streams a video clip off the Internet, and does not involve much computation/processing by the processor

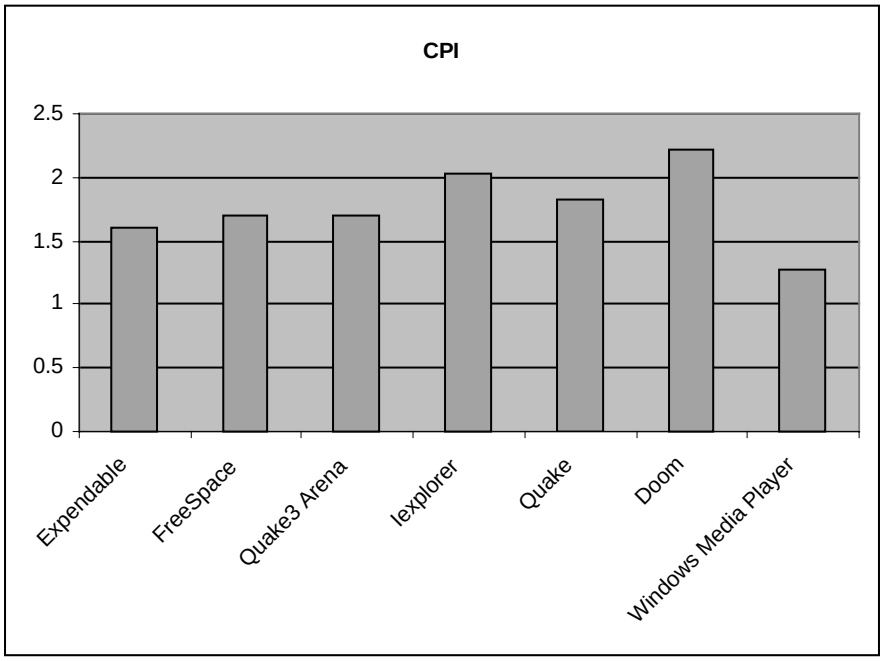


Figure 15: Cycles Per Instruction

itself. This can be inferred from Figures 16 and 17, which show that the Media Player incurs among the least L2 cache misses (which are the most expensive), and also has the fewest number of branches. Figure 19 shows that it also has the fewest resource stalls, thereby giving it a much-improved CPI.

4.2.2.2 Cache Misses

The on-chip cache subsystem of the Pentium III consists of 16kb non-blocking 4-way set associative L1 and L2 caches, with a cacheline size of 32 bytes. The caches employ a write-back mechanism with pseudo-LRU replacement. Figure

16 shows the L1 I- and D-cache misses, and the L2 misses per thousand instructions for the set of benchmarks. The latency of an L1 miss is 4-10 cycles, and that of an L2 miss is about 50 cycles. Thus, L2 misses are really expensive, and often determine benchmark performance. This is borne out in the case of Internet Explorer, which has the maximum number of L2 misses, and therefore, also a high CPI. L1 I-cache misses are generally low except in the case of FreeSpace and Quake 3. We can infer from this that these two games probably have irregular control structure with a lot of control transfers.

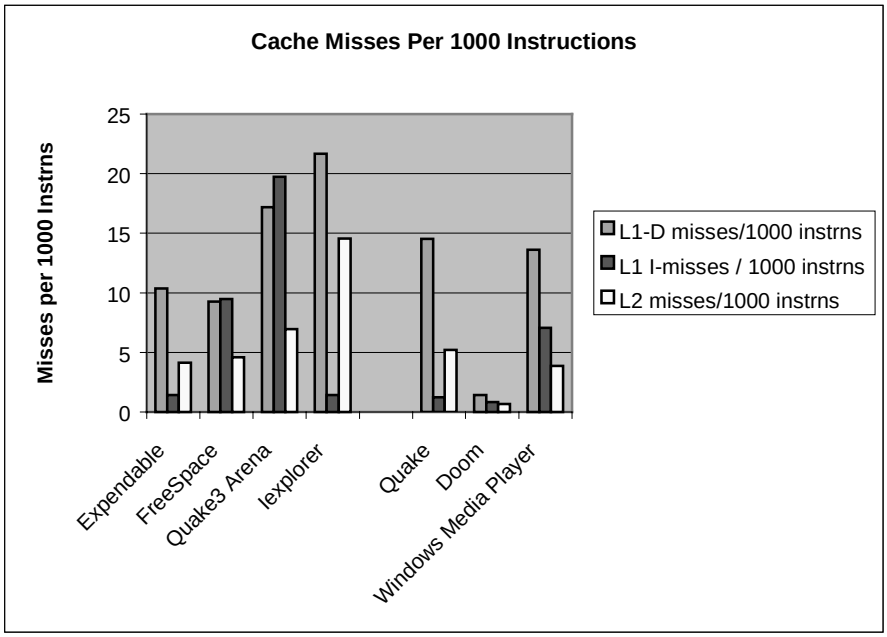


Figure 16: Cache Misses

4.2.2.3 TLB Misses

TLB Miss events in all cases were negligible. Presumably, these events would not have a perceptible effect on overall CPI. TLB misses are therefore not considered hereafter.

4.2.2.4 Branch Prediction

Branch prediction accuracy is critical on a deeply pipelined architecture such as the P6. Outcomes of conditional branches are not known till the execute stage (stage 10), and the minimum branch penalty is therefore 11 cycles. On an average, a μ-op spends about 4 cycles in the reservation station, so a mispredicted branch will typically cause a 15-cycle penalty. If the branch instruction waits in the RS longer, the penalty will be even more. To minimize this penalty, the P6 predicts branches dynamically using Yeh's two-level adaptive training scheme [15]. The P6 has a 512-entry Branch Target Buffer (BTB) which stores branch history as well as the branch target. Branches that hit in the BTB are predicted dynamically; those that miss are predicted statically using a Backward-Taken Forward-Not-Taken scheme. Branch penalties incurred in various cases are summarized in Figure 17.

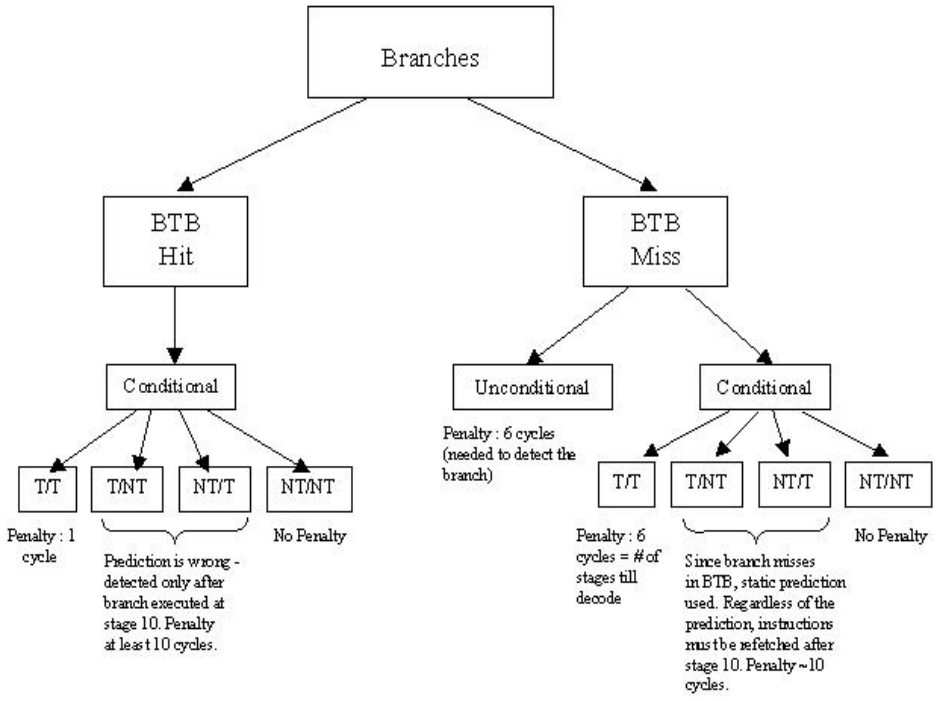


Figure 17: Branch Penalties

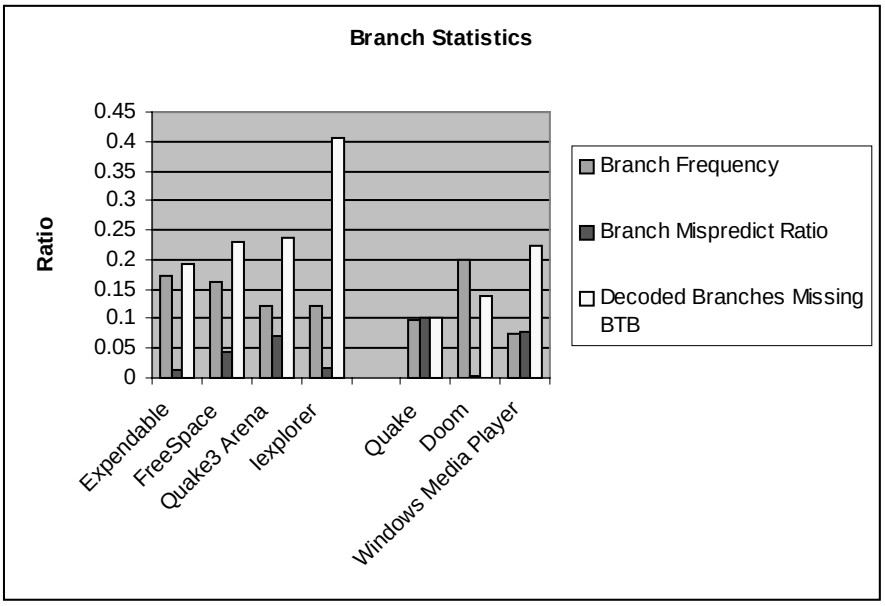


Figure 18: Branch Statistics

Figure 18 shows the branch frequency, the BTB miss ratio, and the branch mispredict ratio over the set of benchmarks. Branch frequency is generally less than 15%. Branch mispredict ratio is less than 10%, which means that most of the branches are correctly predicted, either by the BTB, or statically by the static prediction mechanism. Doom is slightly aberrant among all applications considered, since it has the highest branch frequency, and yet, the lowest branch mispredict ratio. It also has a very low BTB miss ratio. This means that it has a lot of branches in loops that are correctly predicted by the BTB's dynamic prediction mechanism. The branches that miss the BTB are evidently caught by the static prediction scheme and again are predicted correctly. Note that Doom has the lowest branch mispredict ratio as well as the lowest cache miss ratio. Yet, it has the poorest CPI among the benchmarks considered. This implies that there is something else that is eating away performance. This is explained in the next section on Resource Stalls. Explorer has a modest branch frequency (12%), and close to half of these branches miss the BTB.

Evidently, there are a lot of unconditional branches, which, according to Bhandarkar et al [1], are not stored in the BTB.

4.2.2.5 Resource Stalls

Figure 19 shows the number of cycles lost per instruction because of resource stalls and I-fetch stage stalls. The I-fetch stage stalls when there are cache misses, ITLB misses, ITLB faults, or other stalls. Resource stalls occur when resources such as the reorder buffer, the reservation station or memory buffers are full or the functional units are busy. This count also includes stalls arising from branch misprediction recovery.

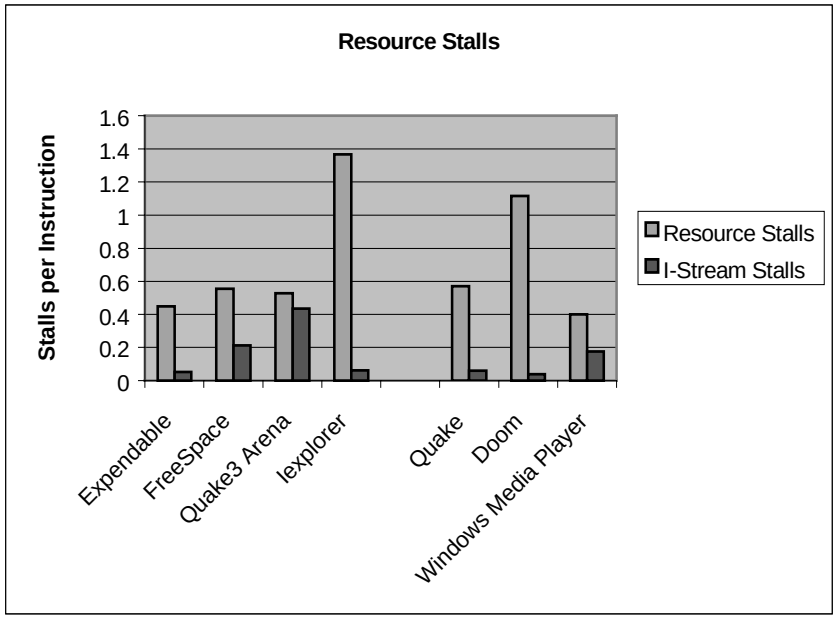


Figure 19: Resource Stalls

The figure shows the primary cause for poor performance in Doom and Explorer: resource stalls. Comparing Figures 15 and 19, it can be seen that the CPI variation is pretty much defined by the variation in the resource stalls. Also note that the I-stream stalls track well with the I-cache misses indicated in Figure 16.

4.2.2.6 Drawing a Correlation Between CPI and Stalls

As the preceding paragraphs have shown, numerous stalls are incurred during execution in an out-of-order processor such as the P6. Intuitively, it may be inferred that the more the cycles lost in various stalls, the more will be the CPI for a particular program. Figure 20 shows a graph with the stall cycles juxtaposed against a graph of the CPI across the set of benchmarks considered. One point to consider here is that the counters give the counts only for the *number* of events, and not for cycles lost due to these events. For instance, the counters tell us only the number of cache misses or branch mispredicts; they do not give any indication of the cycles lost in the events. We have tried to account for the actual stall cycles arising from events using the following considerations:

- L1 cache misses incur a penalty of 5 cycles
- L2 cache misses incur a penalty of 50 cycles
- Deducing the penalty from branch statistics is much more involved because of the elaborate branch prediction scheme on the P6. As shown in Fig. 15, penalties due to branches can be determined from the following :

The P6 provides counts for the following events, assigned letters for simplicity:

K = branches retired

L = mispredicted branches retired

M = taken branches retired

N = taken mispredicted branches retired

O = branch instructions decoded

P = number of BTB misses

Q = cases in which a static prediction is made for the branches

We can deduce the counts for the following events from the above data:

Number of BTB hits = (branch instructions decoded) – (number of BTB misses)
= O – P

Taken correctly predicted = (taken branches retired) – (taken mispred br.retd) branches retired
= M – N

[Note that taken correctly predicted branches need to be considered since these also incur a penalty, which is 1 cycle if the branch hits in the BTB, and around 6 cycles if it misses and has to be statically predicted.]

Unconditional branches = (number of BTB misses) – (number of static preds)
= P – Q

[In deducing this, we note that unconditional branches are not stored in the BTB [1]. Also, all conditional branches that miss in the BTB are statically predicted. Therefore, the number of BTB misses less the number of static predictions made must approximately equal the number of *unconditional* branches encountered during

execution. Each unconditional branch incurs a penalty of around 6 cycles, which is the length of the pipeline stage till decode.]

Thus, the net branch penalty is given by –

Branch Penalty (cycles)	=	Penalty for Mispredicted Branches	+	Penalty for correctly predicted taken branches that miss in the BTB	+	Penalty for correctly predicted taken branches that hit in the BTB	+	Penalty for unconditional branches

$$= L * 12 + (M–N)*(O–P)*1/O + (M–N)*P*6/O + (P–Q)*6$$

Using the above penalties for cache misses and branches, we get the graph shown in Figure 20. It can be seen that the CPI (plotted in Figure 15 and repeated in Figure 21 for convenience) does track the various stalls in the processor. As stall cycles increase, so does the CPI. Note that some discrepancies remain in such an exposition, since there is a lot of overlapped execution that goes on in the machine. For instance, execution can proceed in a resource stall cycle in some other part of the machine. Hence, only an approximate correlation can be drawn between that stalls and the CPI.

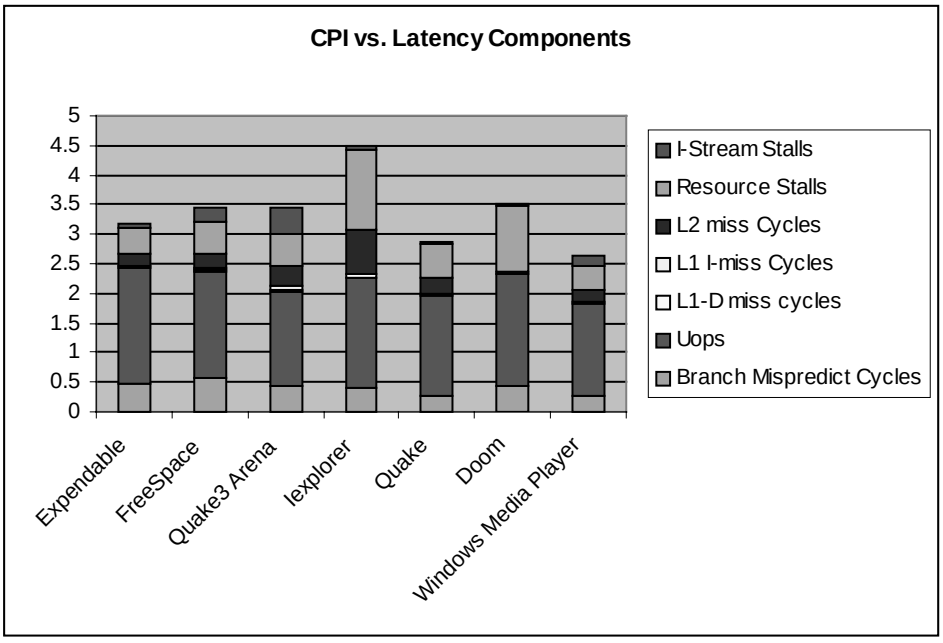


Figure 20: Latency Components

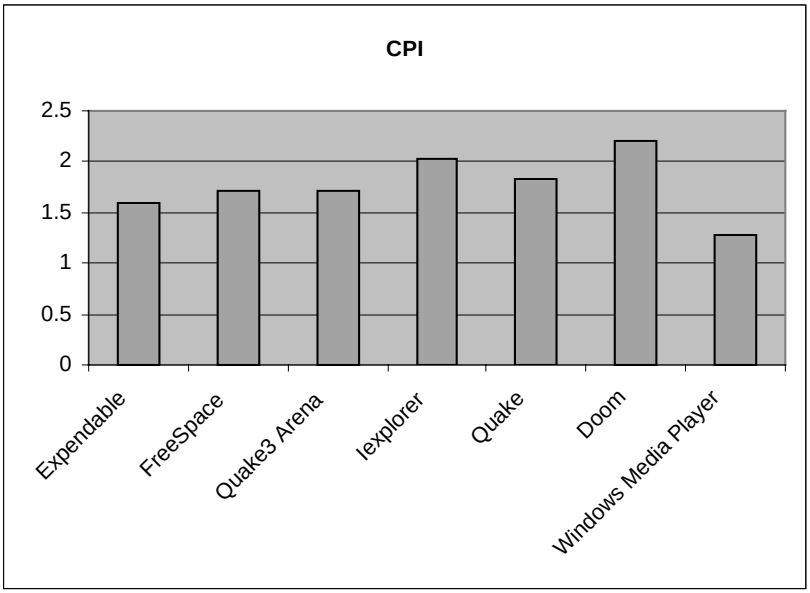


Figure 21: CPI Variation

4.3 COMPARING EXECUTION CHARACTERISTICS ON THE PENTIUM II AND PENTIUM III PROCESSORS

This section compares the performance of the 3D WinBench benchmark on the Pentium II and Pentium III processors. In doing such a comparison, it is essential to ensure that other than the processor, the rest of the system stays the same. While this was not exactly achievable, we have tried to approach this ideal the best we could. We used the two systems with configurations indicated in Table 9:

Pentium II System	**Pentium III System**
Pentium II Xeon at 450 MHz 16kB/16kB L1 I- and D-caches 512kB L2 512MB Main Memory NVidia Riva TNT2 graphics accelerator	Pentium III at 500 MHz 16kB/16kB L1 I- and D-caches 512kB L2 1022MB Main Memory NVidia Riva TNT2 graphics accelerator
+ Xeon processor has full speed L2 – Slower processor, smaller main mem.	– L2 is only half speed + Faster than the PII, large main mem

Table 9: Pentium II and Pentium III System Configurations

The '+' and '–' signs indicate the positive and negative points regarding each system as far as speed is concerned. The differences between the systems were restricted to those between the speeds of the processors and their L2 caches, and the size of the main memory.

As was said earlier, the comparison was done by running the 3D WinBench 2000 suite on the machines. Specifically, we ran the 3D WinMark 2000 and 3D WinBench 2000 Processor Tests on the machines. These tests give results in terms of frames rendered per second and triangles rendered per second, respectively. The results are tabulated in Tables 10 and 11.

Test	Frames	Pentium II (fps)[1]	Pentium III (fps)	Speedup
Speedway	600	5.23	11.1	2.122
Hangar	600	13.6	22.8	1.676
RustValley	876	32.6	33.8	1.036
Canyon	498	24.2	27.5	1.136
Chamber	866	32.8	33	1.006
Stations	424	29.5	31.3	1.061
Islands	504	32.7	38.9	1.189
RaceTrack*	–	–	–	–
Chapel	300	26.3	38.9	1.479
Net Score	–	21.9	26.4	1.205

* RaceTrack could not be run because of limitations of the graphics accelerator card.

Table 10: 3D WinMark 2000 results

Test	Pentium II (Δps)[2]	Pentium III (Δps)	Speedup
Speedway	0.395	0.957	2.422
Hangar	0.367	0.731	1.991
RustValley	0.751	0.83	1.105
Canyon	1.55	2.53	1.632
Chamber	0.623	0.7	1.123
Stations	0.838	0.96	1.145
Islands	0.85	0.972	1.143
RaceTrack	0.883	1.22	1.381
Chapel	0.592	1.33	2.246
Net Score	0.641	0.99	1.544

Table 11: 3D WinMark 2000 Processor Tests Results

It can be seen that the Pentium III performs much better than the Pentium II, especially on the processor tests, where it offers more than 50% speedup. Very

[1] Fps = Frames per second

[2] Δps = Triangles rendered per second

interestingly, when IPC measurements were taken on the two systems, the PII consistently had *higher* IPC values as compared to the PIII. This was most surprising, since the difference in rendering speed for the two systems was large enough to be even visually perceptible – the PIII system had much smoother, more fluid rendering as opposed to the jerky nature of rendering on the PII. We had expected this to imply higher IPC for the PIII machine.

Further analysis of event counts led us to the following conclusion:

The Pentium III uses the 4-wide FP SIMD instructions to process vertex data for the scenes. To a first approximation, therefore, to perform the same computation, the PII will need 4 times as many long-latency FP instructions. Additionally, it will incur 4 times as many loads from memory. Moreover, some of the most commonly used operations in 3D geometry, reciprocal and reciprocal square root, are heavily optimized in the Pentium III to single instructions (RCP and RSQRT) that execute with 2-cycle latencies. The PII would have to resort to the DIV and SQRT instructions that are very costly in terms of cycles. The result is that the code on the Pentium II has many more instructions, which take a much higher number of clocks to execute. Evidently, the increase in number of instructions is more than the increase in number of clocks, so that the net effect is to increase the IPC even with a decrease in performance.

As an example, for the Speedway scene Processor Test, the numbers obtained were as shown in Table 12.

	Pentium III	Pentium II
Instructions	16,386,293,36	35,464,727,10
Clocks	24,113,340,66	47,916,540,29
IPC	0.68	0.74

Table 12: IPC for Speedway

Both processors did the same amount of work, i.e., render the 600 frames comprising the test. The PII required approximately twice the number of clocks as the PIII, and somewhat more than twice the number of instructions. As a result, the IPC on the PII is more than that on the PIII, even though the performance is markedly poorer.

This leads us to the conclusion that IPC is meaningless as an indicator of performance when comparing SSE-based applications across different architectures. The true measure of performance is the frames or triangles rendered per second.

4.3.1 Analysis of Latency Components on 3D WinBench

We now consider the effect of cache misses, branch mispredicts, resource stalls etc. on the overall CPI, in the same sequence of observations as that presented in Figures 15 through 21. The next couple of pages shows the characteristics of the WinBench scenes on the Pentium II and Pentium III systems.

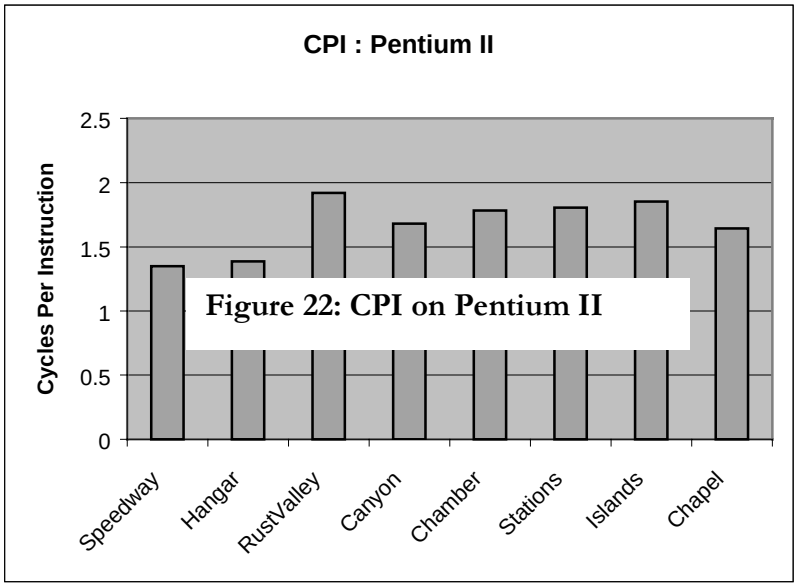


Figure 22: CPI on Pentium II

Figure 23: CPI on Pentium III

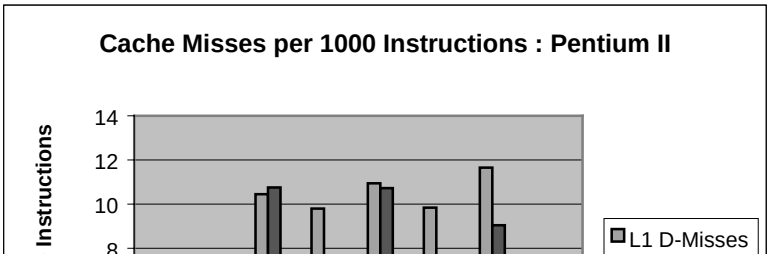


Figure 24: Cache Misses per Thousand Instructions (Pentium II)

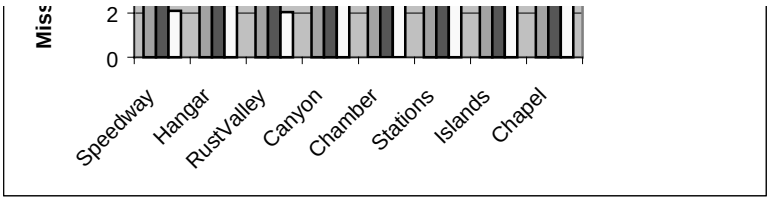


Figure 25: Cache Misses per Thousand Instructions (Pentium III)

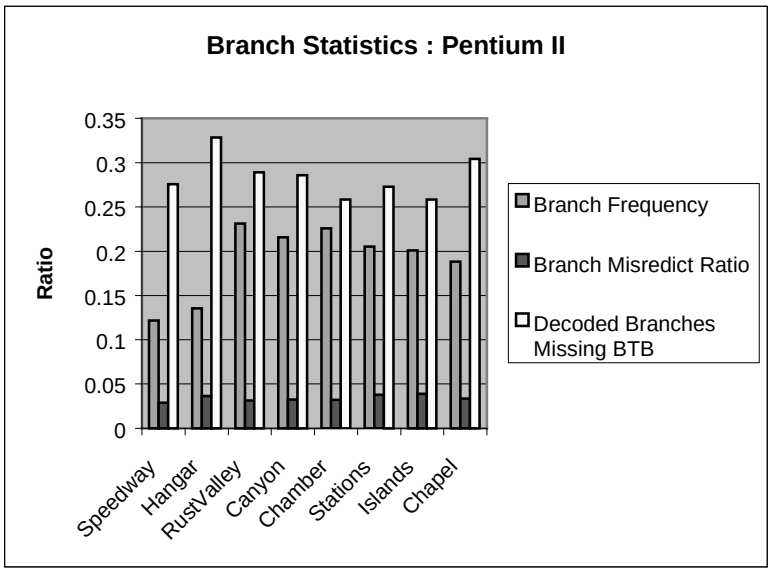


Figure 26: Branch Statistics – Pentium II

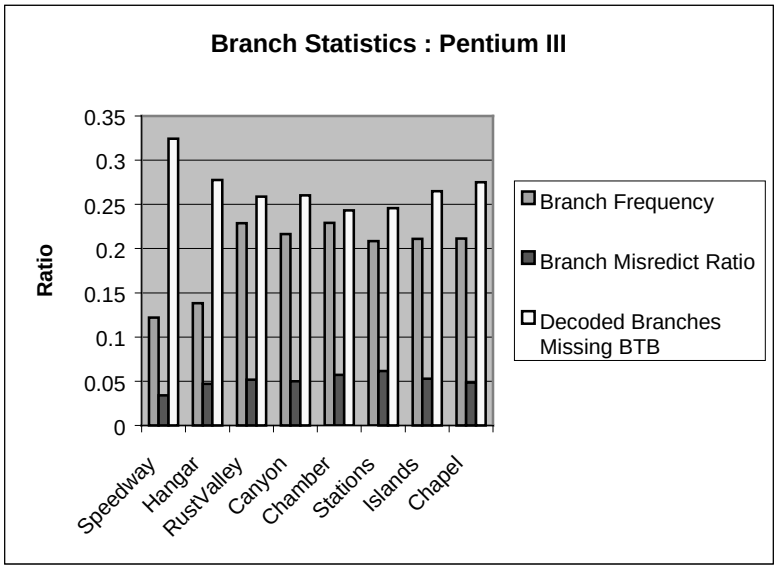


Figure 27: Branch Statistics – Pentium III

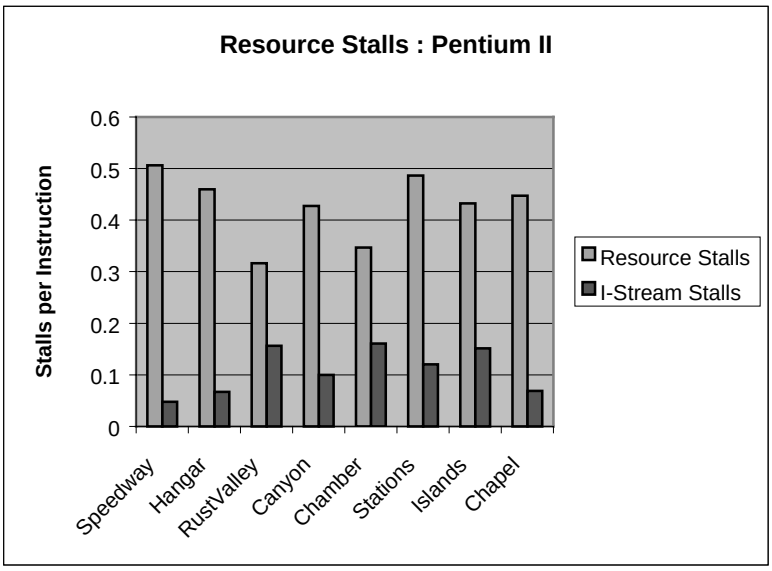


Figure 28: Resource Stalls – Pentium II

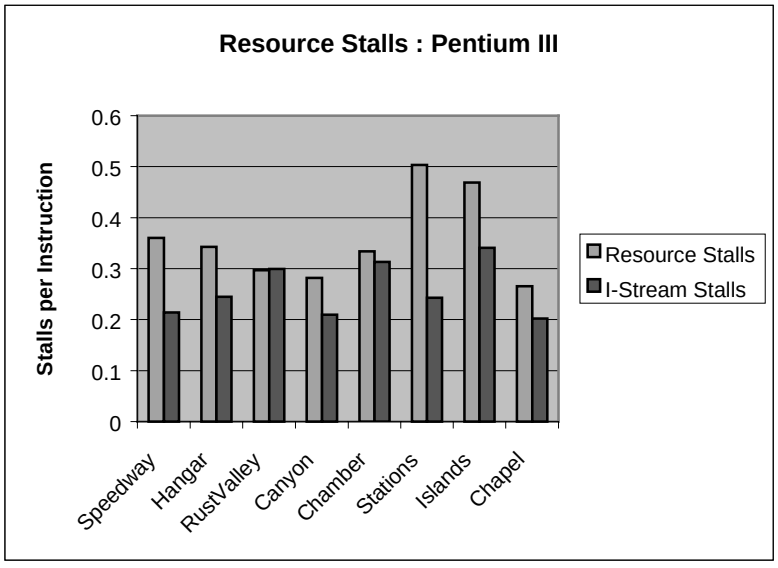


Figure 29: Resource Stalls – Pentium III

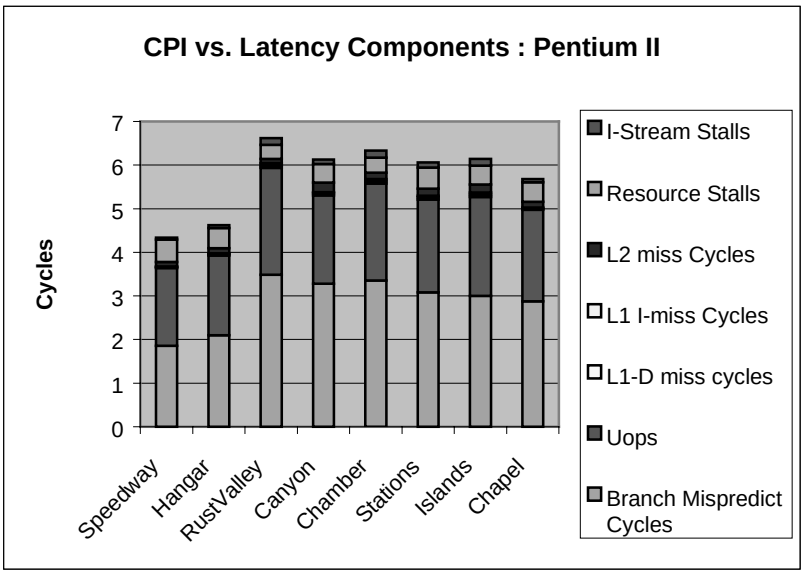


Figure 30: Adding up the Cycles – Pentium II

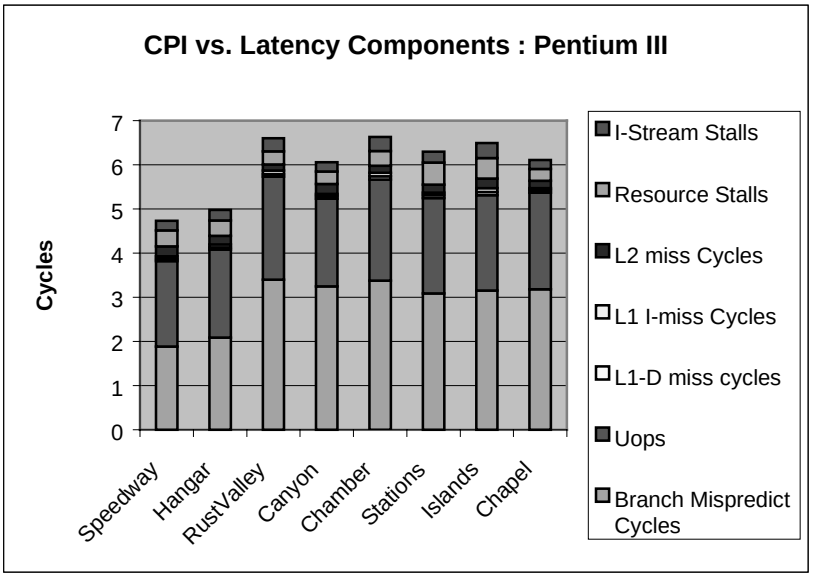


Figure 31: Adding up the Cycles – Pentium III

The readings were taken on the Processor Test for each benchmark. Figures 22 and 23 show the CPIs obtained on the two systems. The CPIs on the Pentium III are consistently *higher* than those on the Pentium II, despite the fact that the benchmarks run much faster on the former processor. The discussion at the beginning of this section shows, however, that this observation is irrelevant. Since the *same amount of computation* has been performed on both systems at the end of each benchmark, a more accurate measure of performance is simply the number of clock cycles counted on each system, or equivalently, the number of triangles rendered per second as explained earlier.

Figures 24 and 25 compare the cache performance of the benchmarks. L2 cache misses are generally low; what is remarkable in this set of benchmarks is that the percentage of L1 cache misses is very high. In particular, the L1 I-cache miss percentage is very high, in most cases *exceeding* the percentage of L1 D-cache misses. This is in stark contrast to the cache behavior seen across the earlier benchmark set (Figure 16), where I-cache misses are minimal. This might suggest a haphazard control structure with a lot of branches. This is indeed borne out by the observation that the overall branch frequency across this benchmark set (about 20%) is much higher than that in the commercial applications (about 12%) seen earlier (Figure 18). Another observation worth noting is that cache misses on the Pentium III are higher than those on the Pentium II in many cases. This could be because of the fact that a lot of prefetches are issued, which miss in the caches and therefore increase the overall miss count.

The branch statistics over the benchmark set are shown in Figures 26 and 27. As mentioned above, the branch frequency is relatively high, around 20% for most of the benchmarks. However, the branch mispredict ratio is low – less than 5% in all cases. This is in spite of a relatively high BTB miss ratio (~25%), which means that

the branches missing the BTB are primarily loop-closing backward branches, or fall-through forward branches, which are both correctly predicted by the static prediction mechanism.

Resource stalls for the benchmarks are compared in Figures 28 and 29. Figure 29 shows that Resource Stalls in the Pentium III are lower than Resource Stalls in the Pentium II; this is probably because of the additional SIMD hardware in the Pentium III that relieves some of the pressure on the existing computation resources. I-stream stalls in the Pentium III are comparable to the Resource Stalls, and are higher than I-stream Stalls in the Pentium II. This is consonant with the earlier observation that the Pentium III has a higher proportion of L1 cache misses than the Pentium II.

Finally, Figures 30 and 31 add up the cycles lost due to the various latency components. Comparing these with Figures 22 and 23 shows that there is a strong correlation between the CPI and the cycles lost due to the latency components.

Chapter 5 : Conclusion

In this study, we evaluated the Streaming SIMD Extensions introduced on the Pentium III processor. We examined the speedup obtained by using the cacheability instructions on a set of kernels. We also characterized the execution characteristics of some representative SSE-enhanced applications. Stall cycles due to various factors, such as cache misses, branch mispredicts, resource starvation etc. were recorded and it was seen that there was reasonable agreement between the CPI and the stalls observed. Finally, we compared performance of the Pentium III and Pentium II processors running the same application with and without enhancements.

The major observations in our study are summarized below:

- Prefetch instructions boost performance more in case of memory-bound applications as compared to CPU-bound ones.
- In kernels such as matrix multiplication, with an irregular memory access pattern, prefetches have to be inserted very carefully. If prefetches are inserted very close to the use of the prefetched data, the prefetch may not have returned the data in memory by the time it is needed. If the prefetches are hoisted much before the data is needed, other data loaded between the prefetch and the use may actually evict the prefetched data because of conflict misses. As such, indiscriminate prefetch may actually hurt performance rather than improve it.
- The prefetch strategy may have to be tailored to the data-set size. A strategy that performs well on one data set may not do so if the data set is changed.

- Commercial applications have a very low percentage of SSE instructions, under 5% in most cases considered.
- Prefetching is apparently not used very effectively in such applications. Our study indicates that most prefetches hit in the caches, which means that the prefetch instructions just consume execution resources without giving any real benefit.
- Our study confirms the conclusions drawn in [1] that it is not possible to derive precise cause-effect relationships in a modern superscalar processor. Only approximate dependencies can be established.
- In comparing performance of 3D applications such as games across architectures like the Pentium II and the Pentium III, IPC is meaningless as an indicator of performance. The only real means of differentiating between the two is by means of other measures such as frames rendered per second (for the entire graphics subsystem) or triangles rendered per second (for just the processor).
- The 3D WinBench tests show a higher percentage of L1 cache misses on the Pentium III compared to the Pentium II.
- Resource stalls on the Pentium III, however, are seen to be lower. This indicates that the added SIMD-FP processing hardware relieves some of the pressure on the existing computation resources.

We hope that this work sheds some light on the execution characteristics of applications enhanced by the Streaming SIMD Extensions. An understanding of how applications actually use the benefits offered by such extensions may be useful for designers of media extensions as well as applications programmers. There are more media extensions on the way, with the SSE-2 enhanced Streaming SIMD Extensions that will be introduced on the Willamette processor from Intel later this

year. This work could be extended to study the next-generation applications using the even more sophisticated ISA media extensions that will no doubt be introduced on future processors.

Appendix

The P6 core has two performance counters. Each counter has an associated event-select register that controls what is counted. Table 13 shows the performance metrics used in this report along with the numerator and denominator events for each metric.

Performance Metric	Numerator Event	Denominator Event
Cycles per Instruction	CPU_CLK_UNHALTED	INST_RETIRED
L1 I cache misses per instruction	L2_IFETCH	INST_RETIRED
L1 D cache misses per instruction	DCU_LINES_IN	INST_RETIRED
L2 cache misses per instruction	L2_LINES_IN	INST_RETIRED
Branch frequency	BR_INST_RETIRED	INST_RETIRED
Branch mispredict ratio	BR_MISS_PRED_RETIRED	BR_INST_RETIRED
Branch taken ratio	BR_TAKEN_RETIRED	BR_INST_RETIRED
BTB miss ratio	BTB_MISSES	BR_INST_DECODED
Resource stalls per instruction	RESOURCE_STALLS	INST_RETIRED
Istalls per instruction	IFU_MEM_STALL	INST_RETIRED
μ-ops per instruction	UOPS_RETIRED	INST_RETIRED

Table 13: P6 Counter-based Performance Metrics

Table 14 gives a description of the events associated with the performance monitoring counters used in this project. More information can be found in [18].

Event Name	Description
CPU_CLK_UNHALTED	Number of cycles during which the processor is not halted.
INST_RETIRED	Number of instructions retired.
L2_IFETCH	Number of L2 Instruction Fetches. This event indicates that a normal instruction fetch was received by the L2. The count only includes cacheable instruction fetches; it does not include uncacheable instruction fetches. It also does not include ITLB miss accesses.
DCU_LINES_IN	Total lines allocated in the DCU.
L2_LINES_IN	Number of lines allocated in the L2.
BR_INST_RETIRED	Number of branch instructions retired.
BR_MISS_PRED_RETIRED	Number of mispredicted branches retired.
BR_TAKEN_RETIRED	Number of taken branches retired.
BR_MISS_PRED_TAKEN_RET	Number of taken mispredicted branches retired.
BR_INST_DECODED	Number of branch instructions decoded.
BTB_MISSES	Number of times that the BTB did not produce a prediction.
BACLEARS	Number of times that a static branch prediction was made, in which the branch decoder decided to make a branch prediction because the BTB did not.
RESOURCE_STALLS	Incremented by 1 during every cycle for which there is a resource-related stall. Includes register-renaming buffer entries, memory buffer entries etc. Does not include stalls due to bus queue full, too many cache misses, etc. Includes stalls arising due to branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.
IFU_MEM_STALL	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults, and other minor stalls.
UOPS_RETIRED	Number of μ-ops retired

Table 14: Event Descriptions

Event Name	Description
EMON_KNI_INST_RETIRED	Number of SSE instructions retired. Based on unit mask, can count either packed+scalar instructions, or scalar instructions only.
EMON_KNI_COMP_INST_RET	Number of SSE computation instructions retired. Based on unit mask, can count either packed+scalar, or scalar only.
EMON_KNI_PREF_ DISPATCHED	Number of SSE prefetch/weakly ordered instructions dispatched. Based on unit mask, can count prefetch NTA, prefetch T1, prefetch T2 or weakly ordered stores.
EMON_KNI_PREF_MISS	Number of prefetches that miss all caches. Can specify type of instruction as above.

Table 14: Event Descriptions (contd.)

References

[1] D. Bhandarkar and J. Ding, "Performance Characterization of the Pentium Pro Processor", Proceedings of High performance Computer Architecture 97, pp. 288-297, Feb 1997.

[2] K. Diefendorff and P. K. Dubey, "How Multimedia Workloads Will Change Processor Design", IEEE Computer Magazine, pp. 43-45, Sep 1997.

[3] R. Bhargava, L. John, B. Evans and R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications", Proceedings of IEEE Micro-31, pp. 37-46, Dec 1998.

[4] D. Talla and L. John, "Execution Characteristics of Multimedia Applications on a Pentium II Processor", Proceedings of 19th IEEE International Performance, Computing, and Communications Conference, Feb 20-22: 2000, Phoenix, Arizona, pp. 516-524.

[5] L. Gwennap, "Intel's MMX Speeds Multimedia", Microprocessor Report, Vol.10, Issue 3, Mar 1996.

[6] K. Diefendorff, P. K. Dubey, R. Hochsprung and H. Scales, "AltiVec Extension to PowerPC Accelerates Media Processing", IEEE Micro, Mar/April 2000.

[7] AMD 3DNow! Website.
http://www.amd.com/products/cpg/3dnow/index.html

[8] S. Thakkar and T. Huff, "The Internet Streaming SIMD Extensions", Intel Technology Journal Q2, 1999.

[9] J. Keshava and V. Pentkovski, "Pentium® III Processor Implementation Tradeoffs", Intel Technology Journal Q2, 1999.

[10] P. Zagacki, D. Buch, E. Hsieh, D. Melaku, V. Pentkovski and Hsien-Hsin Lee, "Architecture of a 3D Software Stack for Peak Pentium® III Processor Performance", Intel Technology Journal Q2, 1999.

[11] D. Papworth, "Tuning the Pentium Pro Microarchitecture", IEEE Micro, April 1996, pp. 8-15.

[12] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design", Microprocessor Report, Vol.9, No.2, Feb 1995, pp. 1-7.

[13] D. Solomon, "Inside Windows NT", Second Edition, Microsoft Press, 1998.

[14] P. Viscarola and T. Mason, "Windows NT Device Driver Development", First Edition, Macmillan Technical Publishing, 1998.

[15] Tse-Yu Yeh and Y. Patt, "Adaptive Two-Level Branch Prediction", Proceedings of the 24th International Symposium and Workshop on Microarchitecture, Albuquerque, Nov 1991.

[16] Ziff-Davis Benchmark Operation (ZDBOp) 3D WinBench Website.
http://www.zdbop.com
http://www.3dwinbench.com

[17] MetaStream Viewer Website
http://www.metastream.com/mts2.html

[18] Intel Architecture Software Developer's Manual, Vol.3: System Programming Guide.

Vita

Vikram Uday Godbole was born in Pune, India on November 6, 1976, the son of Aruna Uday Godbole and Uday Shridhar Godbole. He received the degree of Bachelor of Engineering in Electronics and Telecommunications from the University of Pune in August 1998. In September 1998, he entered The Graduate School of Engineering at the University of Texas at Austin.

Permanent address: 300 E30th #301
Austin, TX 78705.

This report was typed by the author.