TAXI: Trace Analysis for X86 Interpretation

Stevan Vlaovic
Sun Microsystems
4100 Network Circle
Santa Clara, CA 95054
Stevan.Vlaovic@sun.com

Edward S. Davidson
University of Michigan
Advanced Computer Architecture Laboratory
Ann Arbor, MI 48109-2122
Davidson@eecs.umich.edu

Abstract

*Although x86 processors have been around for a long time and are the most ubiquitous processors in the world, the amount of academic research regarding details of their performance has been minimal. Here, we introduce an x86 simulation environment, called TAXI (**T**race **A**nalysis for **X**86 **I**nterpretation), and use it to present some results for eight Win32 applications. In this paper, we explain the design and implementation of TAXI.*

1. Introduction

Interest in applications (beyond simply SPEC) has been increasing within the computer architecture community. In this paper, we introduce a simulation environment that allows us to gauge the performance of some common desktop applications that run on x86 platforms. Previously, the main problem with x86 results is that detailed performance data could not be gathered, other than by gathering statistics of high level events[4,5]. Current offerings from Intel, AMD and others have hardware counters that enable the counting of certain architectural events, while others use annotated binaries to sample events. Although counting certain events does provide insights into performance, the main obstacle to gathering x86 *performance* data is that current x86 processors decompose the x86 instructions into smaller operations, called μops and hide the μop level from the user. In order to depict microengine performance at the μop level, this mapping would have to be implemented anew in the simulator for all the instructions and addressing modes that appear in the applications of interest. We have developed such a simulator, and discuss its design in the paper.

In order to achieve the goal of obtaining performance related information about x86 processors, in our approach there are two different aspects that we have to consider. The first is trace collection, which we discuss in detail in Section 2.0, and the second is the timing simulation, which we present in detail in Section 2.2.

2. Software Emulation

To perform cycle accurate simulation, either the entire simulation, including applications, processor state, and timing can be modeled in software, or a trace can be gathered by hardware or software means. This trace can later be fed into (another) software simulator that tracks dependencies and resource usage (and hence timing) but does not require state information; this is the approach that we took with TAXI.

We chose software over hardware trace collection as it allows for more flexibility in the type of data that we can collect and the systems that we can model, and we are not limited by the size of hardware buffers. Software emulation by itself does not provide microarchitectural performance data. This requires (cycle-approximate) *simulation* of the microarchitecture under study. Where *emulation* is concerned with the correctness of the system under study, *simulation* involves modeling only the characteristics that are of interest, and then gathering accurate information about those characteristics of the simulated system. Simulators usually have both of these components, a *functional simulator* which performs emulation by tracking the state of the system, and a *performance simulator* which keeps track of dependencies in the microarchitecture and provides detailed timing information. These two components can also be separated, by having the functional simulator (or emulator) create traces of program execution. These traces can then be fed into the performance simulator to gather the detailed timing data. The advantage of this separation is in the ability to directly compare the performance of different microarchitecture implementation models of the same instruction set architecture by using the same trace repeatedly. The disadvantage is the space required to store the trace could potentially be very large, depending on the type of information collected. Although this tracing methodology did create large traces, it allowed for trace reuse over multiple models.

In order to implement an x86 performance simulator, the traces had to be rich with information, as presented in Section 2.1. Additionally, the capability of adding value prediction in the future necessitated that values be associated with registers and loads and stores (so that a predicted value could be validated). In Section 3.0, we introduce our simulation infrastructure, called **T**race **A**nalysis for **X**86 **I**nterpretation, or TAXI. There are two primary components of TAXI: the functional simulation (Bochs portion), and the cycle-accurate performance simulation (SimpleScalar portion).

2.1 Bochs

Rather than starting from scratch, we chose to use an open-source Pentium emulator called Bochs[3] available from MandrakeSoft. Bochs, developed by Kevin Lawton, runs on most platforms including Linux/x86 and Linux/PPC. Bochs has been developed in an open "bazaar" style since its inception in 1994 and has involved a few hundred contributors. Since the purchase of Bochs by MandrakeSoft, it has been committed to an Open-Source license (LGPL).

The advantage of using Bochs is that much of the hardware modeling is already provided. Bochs provides emulation for devices such as a keyboard, mouse, hard drive, floppy drive, and a VGA compatible monitor. In fact, it models not just the CPU, but the entire platform in enough detail (after modification) to support the execution of a complete operating system and the applications that run on it. Currently, we are using out-of-the-box Windows NT 4.0 (Build 1381, Service Pack 5) as the operating system for our functional simulator. Our modified Bochs emulator runs as a user-level process on a standard PC (under Linux or Windows NT) and models the platform components completely in software.

Although Bochs proved to be a good starting point, it was by no means complete; a few device drivers had to be added. For example, a sound card device driver had to be added before our Winamp application would run on our functional simulator.

Bochs requires a hard drive, which is similar to any other file. After creating a hard drive image, a floppy image that contains bootable kernels are required to "boot" our functional simulator. The hard drive image then needs to be "formatted", and the operating system installed with the correct hardware settings configured (i.e. 84-key keyboard, VGA graphics, Microsoft mouse, etc.). Although we chose to use Windows NT as our guest operating system, we could have just as easily used variants of Unix (that run on x86 platforms) as well. In fact, since there are much less complicated licensing issues with certain Unix derivatives, the Bochs website contains premade bootable Unix hard drive images. By simply directing Bochs to the proper hard drive, any number of different environments can be simulated.

Since Bochs is generally used as a testbed for developing device drivers (and providing a way to run old DOS games) a portion of the code is untested. Because the applications that we chose to run are more complex than those previously used, a few problems were encountered: the VGA device not being able to handle certain modes, an error in the physical page access (wrap around for the end of memory), and other such anomalies common with untested code.

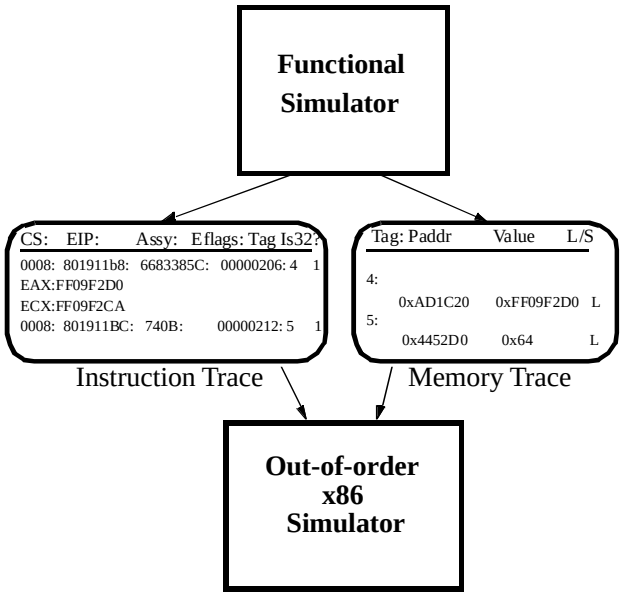

Figure 1. Interaction of functional simulator, out-of-order simulator, and the trace files

Bochs also had to be modified to create a suitable interface to the performance simulator. Figure 1 shows the interaction between the functional simulator and the performance simulator. In this version of TAXI, Bochs was used to generate a trace file as described above, while later revisions can simply pipe the trace to the performance simulator. As seen in the middle left rounded box of Figure 1, our instruction trace contains the following data:

- Segment Register (CS)
- Effective Instruction Pointer (EIP or PC)
- Assembly of the Instruction (Assy)
- Eflags Register Contents
- Tag
- 32-bit code flag

This is shown in the first line of the lower left box, which represents one x86 instruction. The results of executing this instruction are shown by the register contents following the instruction. In this example, the segment register value (CS) is 0x0008, the EIP is 0x801911B8, the assembly is 0x6683385C, the Eflags

register is 0x206, the tag is 4, and instruction is a 32-bit instruction. Even though our performance simulator only tracks dependencies (as opposed to actually performing the computation), the inclusion of state (register values) was added to facilitate future enhancements (such as value prediction or wrong path execution). Anytime a register changes value, we output that value to the instruction trace file. The next line in the instruction trace in Figure 1 shows that the EAX register contents changes to 0xFF09F2D0 as a result of the execution of this instruction.

For the performance simulation, the data address of each load and store is also required. We create a second trace file (i.e. each run of an application produces two traces, one for all the instructions, and another that has all the load/store addresses) that contains the following information which is needed to associate each address with a particular load/store instruction:

- Tag
- Physical Address
- Value
- Load or Store

The *Tag* is used to associate the address with a particular instruction in the first trace. Since each instruction is tagged, by inspecting the memory trace file all the loads and stores for that instruction (*tag*) can be determined. In the example of Figure 1, the first instruction is tagged with the number 4, the memory trace file is searched sequentially until the *tag* 4 is found. The resulting *physical address*, 0xAD1C20, is then used to model the access to memory.

2.2 SimpleScalar

Rather than start from scratch, we chose to start with an existing out-of-order simulator, SimpleScalar, developed by Austin and Burger[1]. SimpleScalar is the most flexible existing package with open source code, which made it most suitable for our needs.

SimpleScalar performs fast, flexible, and reasonably accurate simulations of modern processors that implement the SimpleScalar Instruction Set Architecture (ISA). Although there are no production processors which use this ISA, it is based on the MIPS instruction set. The SimpleScalar ISA, called PISA, is relatively easy to modify and extend. The performance simulator portion of SimpleScalar has been aggressively tuned for high speed simulation, making it easier to run larger, more realistic applications.

Although SimpleScalar is a good starting point, currently it cannot directly handle x86 applications. However, starting with the Pentium Pro, Intel and most x86 processor manufacturers began dynamically translating x86 instructions into (computationally) smaller, more RISC-like instructions (µops), which are then directly executed by the microarchitecture. These µops are not unlike PISA instructions. For instance, a memory-register ADD can be decomposed into the following PISA instructions:

```
                          LW Temp1, addr
ADD addr, REG  ------>    ADD Temp1, Temp1, REG
                          SW Temp1, addr
```

Since Intel and other x86 processor manufacturers who use µops do not publish the actual decomposition, we have developed our own µop mapping which is close to Intel's µop mapping. By using the performance counters in the Pentium II, we can measure the average number of µops per instruction, which is within 10% over the applications we studied. We provide a decomposition of x86 instructions into PISA instructions (µops) for all addressing modes and for roughly 150 x86 instructions, which is sufficient for all our applications.

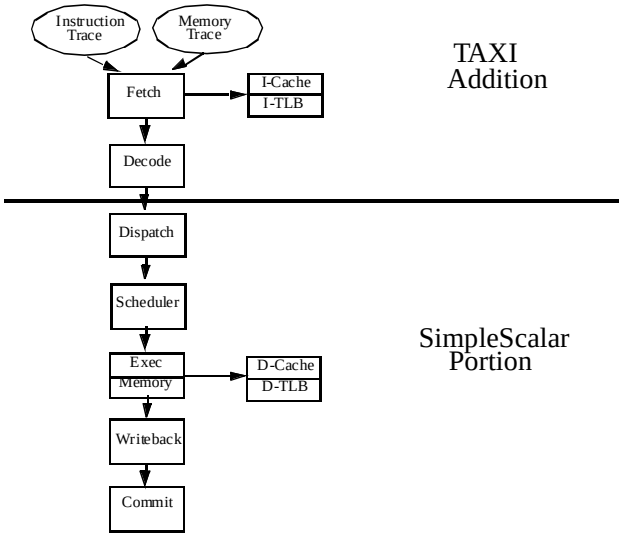

Figure 2. Functional pipeline for x86-outorder, TAXI addition is the front end of the pipeline, SimpleScalar is the back end

In addition to creating the µop breakdown, significant changes had to be made to the front end of SimpleScalar's out-of-order pipeline model to obtain the pipeline sketched in Figure 2. Both the instruction and memory reference trace files, (as shown in Figure 1), are inserted into the performance simulator at the Fetch stage. The next two subsections describe the Fetch and Decode functional stages of the out-of-order model. Note that we can arbitrarily associate any number of actual pipeline stages to each of these functional stages. For instance, in our Pentium III model, Fetch takes 3 pipeline stages, represented by `IFU1`, `IFU2`, and `IFU3`.

2.2.1 Fetch

In the fetch stage of the TAXI performance simulator, the first task is to read in x86 instructions from the instruction trace (previously generated by the x86 emulator). Next, using the tag information, the memory addresses (if any) associated with the current instruction are read in from the memory trace. These instructions are then converted into cache line addresses so that the *Streaming Buffer* can be represented. The Streaming Buffer in any generic x86 processor holds up to *n* cache lines, aligned on cache line boundaries. The Streaming Buffer holds the raw instruction stream aligned on cache line boundaries while the processor's front end breaks down the stream into instructions to be fed into the x86 to μop decoders. In the Intel P6 architecture family the Streaming Buffer holds one cache line [6].

The Streaming Buffer is further broken down into *blocks*, with one block transferred per cycle to the instruction to μop decoders. For instance, if the cache line size is 32 bytes, and the internal decode width of the processor is 16 bytes, there are 2 blocks per cache line (i.e. 2 cycles to decode one cache line). Since x86 instructions are unaligned, yet the Streaming Buffer is aligned on cache boundaries, the number of instructions that can fit into an *n* line Streaming Buffer must be calculated, such that the number of instruction cache accesses can be correctly modeled.

Using the address of the instruction and its length, the number of instructions that fit into the decode width can be determined. For instance, a decode bandwidth of 16 bytes can handle eight consecutive 2 byte instructions, or alternatively three 5 byte instructions (with one byte of wasted bandwidth if the next consecutive instruction is more than 1 byte). (Note that the first instruction is *always* aligned.) This is exactly how the actual hardware would partition the buffer into x86 instructions and access the L1 instruction cache.

The instructions are converted into cache and TLB accesses only if the instructions do not reside in the Streaming Buffer. There is no need for either translation or cache line fetch for a cache line that is already present in the internal buffer of the processor.

Since our implementation uses instruction traces, it is difficult to model wrong path execution of an out-of-order machine. Rather than keeping wrong path information in the trace (this makes the trace much too large), we mark the last μop of an x86 branch instruction that is incorrectly predicted. The front of the pipeline is then stalled (no more instructions are read in from the trace); as soon as the marked μop its executed, the front of the pipeline is allowed to resume reading correct path instructions from the trace. Although this does not capture second order effects (e.g. cache and TLB state modified by the wrong path execution), it does correctly model the nominal recovery latency corresponding to a branch misprediction.

Because we have values embedded within the trace, we can actually model wrong path execution. By including an emulation engine within the performance simulator, we can traverse down incorrect paths. (The caveat here is that load values may not be known if we have not seen this memory access previously.) With this framework, we could measure the performance impact of modeling speculative execution versus handling mispredictions in the manner that we do with TAXI.

2.2.2 Decode

In the Intel and AMD current x86 architectures, three types of decoders are used to break down x86 instructions into μops. A *simple* decoder can decode all instructions that break down into a single μop. A *complex* decoder can decode the instructions that decompose into more than one, but less than five μops. Finally, a MicroInstruction Sequencer (MIS) is needed for all instructions that decompose into five or more μops. TAXI models the simple and complex decoders and the MIS.

In the decode stage we need to decode the x86 instructions and put them into a Decoded Instruction Queue (DIQ) where the resulting μops are processed like any other out-of-order load/store processor that has fixed-length instructions. This stage continues decoding (putting instructions into the DIQ) as long as the following holds:

- available width from the Streaming Buffer
- available simple and complex decoders
- available temporary registers
- available space in the Decoded Instruction Queue

It is essential for x86 instructions that decompose into many μops that the DIQ accept partial μops lists (only a fraction of the μops from each instruction). For instance, the `bswap` operation requires eleven μops in our decomposition, and if the DIQ can hold a maximum of six μops, then an all or nothing policy would deadlock the processor. Obviously, the TAXI implementation allows for partial μop decomposition.

2.2.3 Miscellaneous Additions

There are other changes that had to be made to SimpleScalar in order to get it to work correctly within the TAXI infrastructure. The load and store addresses that are stored in the Memory Reference trace must be associated with each μop when the x86 instruction is decoded. Then, in the *Exec/Memory* phase of SimpleScalar shown in Figure 2, these addresses are used when accessing the D-cache and D-TLB. Since SimpleScalar contains both a functional and a performance simulator, it generates the

necessary data reference addresses (whereas TAXI gets the addresses from the memory reference trace).

An interesting and somewhat limiting aspect of the x86 architecture is the number of architected registers: only 8 general purpose registers are architected. Since having only 8 registers would be a severe limitation, most x86 architecture implementations that generate μops employ non-architected registers, i.e. registers that are not visible to the software and are managed entirely by hardware. These registers are used for loads and stores and other instructions that require temporaries (e.g a memory-register add which decomposes into a load-add-store μop sequence). In the TAXI infrastructure, the number of these non-architected registers can be varied to gauge the impact of increasing the register space.

The floating point register space in the x86 architecture is defined as an 8 entry stack. Since this, too, is somewhat restrictive, many hardware implementations flatten out this register space and allow for more non-architected floating point registers, which is the approach taken in TAXI.

3. TAXI

The combination of the modified Bochs and the augmented SimpleScalar comprise the heart of TAXI. However, for doing simple cache or branch prediction studies, then running the out-of-order simulator might be unnecessary. For this reason, the TAXI infrastructure includes some specialized analysis tools, such as cache, branch predictor, and Branch Target Buffer (BTB) simulators.

For simple cache studies, the dinero-IV cache simulator by Edler and Hill[2] was modified to accept our trace files. For the instruction cache, the instruction trace is used as input, with the pertinent information extracted being the Effective Instruction Pointer (EIP) and the instruction length. For the data cache, the address can be selected to be either the virtual address or the physical address with the loads corresponding to reads, stores corresponding to writes. We used the physical address to access the data cache, using 32 MByte (configurable in Bochs) of main memory for the functional simulator when creating the trace files.

Branch prediction is another performance aspect that is useful to model without the out-of-order simulation overhead. To do our branch prediction studies, the branch prediction mechanism of SimpleScalar was modified to accept our x86 instruction trace. For some of our BTB studies, only the BTB portion of the branch predictor was used. The impact of architectural changes to these resources can be rapidly discovered with these three simplified, specialized tools.

To better understand the run-time behavior of applications, it is sometimes useful to target the code that is most frequently used. By having a correlation between function names and absolute function addresses, we can target those functions that cause the greatest performance detriment. To do this, the mapping of functions to virtual memory is required. Since Win32 (i.e. Windows) applications group common functions into Dynamically Linked Libraries (DLLs), the locations of DLLs and their functions in virtual memory must be known. The only way to gather this information is in the virtual environment. To do this we used a tool called `pwalk.exe` (included in the Win32 Software Development Kit) which is run first, then loads the executable under study. The process is then *walked*, giving the loaded virtual memory address of every DLL that is loaded. Next, to get the addresses of the functions within a DLL, the export table can be extracted using `objdump.exe` (also included in the Win32 SDK). Note that this table just gives the address of exported functions (and variables), not every function.

To get the address of every function, a `debug` version of Windows NT has to be used; this version comes with associated `.SYM` files. The `.SYM` files contain the addresses of all functions and variables in the DLL or executable. Because the instruction trace contains the addresses of each call, it is possible to count the number of times that each particular function gets called within the execution trace. Additionally, if a particular sequence of calls is of interest, then this can also be determined from the instruction trace. Essentially, these `.SYM` file tables, which have absolute function addresses, just correlate instruction addresses with actual function names, much as a debugger does. This correlation makes gathering information and gaining insight into the operation of an application much more straightforward.

For our studies, we did not require complete information at this granularity (and we wanted to run the actual OS, not the debug version as the debug version would have a larger footprint and run slightly slower); thus, the DLL mapping for each run of the application, as obtained from `pwalk.exe`, was sufficient for our purposes. Note that these mappings (either the object or the function) are *not* required to use TAXI: these merely provide useful information regarding how the Win32 application behaves.

Lastly, a generic trace reader is also included in the TAXI infrastructure. This allows for rapid simulation of other architectural features that are not included in the TAXI infrastructure (e.g. L2 instruction cache performance). Eventually, we could merge Bochs with the out-of-order simulator we developed so that collecting traces would not be necessary. This would allow for even greater flexibility and the ability to model wrong-path

information and much longer instruction sequences than we currently are able to perform with traces.

4. Applications

We have chosen eight popular Windows NT applications: *Id's Doom, FileMaker Pro 5.0, Microsoft Explorer 5.0, Microsoft Visual Studio 5.0, Netscape 6.0, RealPlayer 8.0, Winamp 2.72, and Winzip 8.0*. *Doom* is one of the early first-person type combat games and is available as shareware. The run of Doom included recording a session of a Doom game, and then replaying it on the simulator. *FileMaker Pro 5.0* is a database application that allows users to easily share their data over the internet. This run consisted of performing multiple searches and sorts on a 3,000 entry database. *Explorer 5.0* is Microsoft's web browser; our input is a set of three `.htm` pages. The first is the CNN web page, the second is an ESPN web page, and the third is University of Michigan's EECS homepage. *Microsoft Visual Studio 5.0* (MsDev) is a code development environment, with 5.0 being the previous release. Our run of Visual Studio involved the compilation of *go* from the SPEC95 benchmark suite. *Netscape 6.0* is another popular web browser; the same web pages that were loaded on Explorer were also used for Netscape. *RealPlayer 8.0* is a video player that can be used to play a number of different video formats with the second episode of SouthPark being used as its data input. *Winamp 2.72* is the latest release of a popular mp3 player; its input was "Cool Down Daddy" by Jellyroll. *Winzip 8.0* is a compression and decompression engine that can handle multiple formats. Our run of Winzip entailed compressing the source files from go (from SPEC95).

Table 1 highlights some dynamic characteristics of the applications that were obtained using TAXI tools. In the first column, the number (dynamic count) of x86 instructions in each trace is shown. The average number of μops/instruction is presented in the second column. Note that the average of these traces (weighted average where each trace is weighted by the length of each trace) decomposes into 621 million μops (420 * 1.48). The third and fourth columns show the number of instructions per store and load, respectively (e.g. on average 1 of every 4.80 instructions is a store instruction). In the Instructions per Branch column, LOOP and REP instructions as well as calls, returns, jumps, etc. are considered branches, and are therefore included in this number. These ratios imply that 100 instructions would on average contain 21 Stores, 37 Loads, 20 Branches and 22 instructions of other types; however 148 μops would on average still contain 21 stores, 37 loads, and 20 branches, and thus 70 μops of other types, many of which either arise from load, store, and branch instructions themselves or are otherwise in direct support of loads, stores, and branches.

	Insts. ($x10^6$)	μops/ Inst	Insts/ Stores	Insts/ Loads	Inst/ Brnch
Doom	388	1.52	4.61	2.64	5.40
Explorer	396	1.50	4.53	2.55	4.77
FileMaker	447	1.49	4.61	2.85	5.04
MsDev	469	1.46	4.94	2.63	4.07
Netscape	377	1.52	4.53	2.65	4.33
RealVideo	522	1.44	5.44	2.76	4.63
Winamp	456	1.47	4.11	2.31	6.88
Winzip	302	1.44	5.74	3.05	5.63
Average	420	1.48	4.80	2.67	5.07

Table 1. Application trace characteristics

5. Validation

To get some insight into how accurately TAXI models actual performance, we used the performance counters on the Pentium II and ran the same applications with the same input sets that were used with our TAXI Pentium II model based on [6]. Note that although we are mostly faithful to the implementation described in [6], there are still details of the microarchitecture that are not described. Table 2 shows the parameters for our model of the Pentium II; an * indicates where we may diverge from the real implementation. This information gap grows even larger when it comes to the Pentium 4; our Pentium 4 model contains even more assumptions about the microarchitecture. Our baseline "real" system is a 300 MHz Pentium II processor.

Parameter	Value
Physical Registers*	64-INT, 64-FP
Streaming Buffer Size	32 bytes (1 cache line)
Decode width	16 bytes
# Complex Decoders	1
# Simple Decoders	2
Decoded Instruction Queue*	8 μops
μop Decode Width*	4 μops
μop Issue Width*	4 μops
μop Commit Width*	4 μops
ROB*	40 μop entries
Branch Predictor*	GAg, 512 entry
BTB	512 entry 4-way
Return Address Stack	8 entries
L1 D-cache	16KB 4-way
L1 I-cache	16KB 4-way
L2 Unified	256KB 4-way, 6 cycle latency
TLB size	32 Instruction/64 Data
Memory*	120 cycle latency

Table 2. Pentium II model parameters

One main problem is that operating system behavior is not deterministic (due to service scheduling, etc.), but

taking the average of five runs reduces this effect. These results shown in Figure 3 are given in terms of Cycles Per Instruction (CPI). There are a number of possible reasons for the discrepancies between the Real Pentium II and the TAXI model. First, since operating system activity is included, variations in scheduling operating services can affect what is actually executed during a given execution run. This leads into the another potential problem: it is difficult to start the counters timing from the exact same spot (or even run them for the same duration); thus the counters may be capturing information from different portions of the program. The other difficulty is that the µop mappings are different. Although, on average, TAXI is within 10% (of the µops per instruction ratio), certain portions of applications can make particular use of inaccurate instruction to µop mappings. The greatest difference between the TAXI model and the measured Pentium II is seen in the application RealVideo at 23%, with the average difference being 7% across the application suite studied.

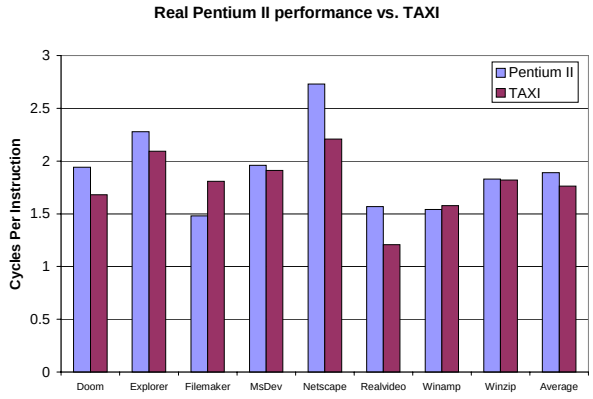

Figure 3. "Real" Pentium II results compared to results obtained using the TAXI model

Additionally, a test kernel was run on both a Real Pentium II and the TAXI model. This test kernel is comprised of only two instructions, a register subtract and a branch back to the subtract (i.e. they comprise a simple loop), which lie in the same cache line. Once the cache line is loaded into the processor's Streaming Buffer, there are no cache accesses. The results in terms of Cycles Per µop (CPµ) are shown in Table 2. The baseline results are the average of five runs of 10 seconds each. The difference in performance is roughly 6%.

System	CPµ
Baseline	1.50
TAXI Pentium II Model	1.59

Table 2. Test kernel measurement

While in our model, we tried to be faithful to Intel's implementation of the Pentium II, certain aspects of the microarchitecture are not published. In particular, µop mapping details are not disclosed, and the mapping in our model, although reasonable, is no doubt somewhat different. Furthermore, the simplified processor model simulation makes it difficult to capture processor performance exactly. However, we feel that TAXI is sufficiently powerful to enable realistic design evaluations and trade-offs to be made regarding current and future x86 processors.

6. Conclusions

The x86 instruction set architecture is much different than other, more conventional ISAs. Its variable length instructions, numerous addressing modes, and restricted architecture state make the microarchitecture implementation much more complex than for RISC ISAs, especially where high performance is important. With the information provided by TAXI, we can determine the individual impact that individual components have on overall performance and focus processor improvement efforts on those components that have the highest impact. We look forward to exploiting this new TAXI infrastructure to carry out further studies on x86 architectures, applications, and innovations.

7. References

[1] D. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar ToolSet," University of Wisconsin-Madison, Computer Sciences Department, Technical Report CS-TR-1308, July 1996.

[2] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," http://www.neci.nj.nec.com/homepages/edler/d4

[3] K. Lawton, "Welcome to the Bochs x86 PC Emulation Software Home Page!" http://www.bochs.com

[4] D. C. Lee, P. J. Crowley, J-L Baer, T. E. Anderson, and B. N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT," Proceedings of the 24th International Symposium on Computer Architecture, pp. 27-38, IEEE, 1997.

[5] S.E. Perl and R.L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," Digital Systems Research Center Research Report, RR-146, April 1997.

[6] T. Shanley, Pentium Pro and Pentium II System Architecture, Addison-Wesley, September 1999.