The Effects of the x86 ISA on the Front End: Where have all the cycles gone?

Stevan Vlaovic and Edward S. Davidson

Advanced Computer Architecture Lab

The University of Michigan

{vlaovic, 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 will introduce x86 simulation environment, which we call **T**race **A**nalysis for **X**86 **I**nterpretation, or TAXI, and use it to discuss the differences between current x86 processors and other processors, and present some performance results of eight Win32 applications. By utilizing TAXI, we can attribute performance bottlenecks to those components of the microarchitecture that cause the most performance degradation. We look at 8 aspects of front-end that can contribute to performance loss; then based on this information, we introduce an improvement that yields 17% speedup in overall execution time.*

1.0 Introduction

Interest in commercial applications has been increasing within the computer architecture community. In this paper, we present 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 statistical sampling of high level events[7,8]. 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 sampling 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 have used it in this paper to illustrate the impact that the x86 instructions have on the front-end of the processor.

All x86 processors that decompose instructions into μops follow the same general algorithm, as seen in [12]. First, instructions are fetched from the instruction cache and put into a Streaming Buffer which is just a container to hold the cache-line(s) while the processor parses it into instructions. Next, since the instructions are not aligned, the instruction boundaries need to be determined. Finally, the instructions are moved to the instruction-to-μop decoders and decoded, and the μops are put into the decoded instruction queue. *Simple* x86 instructions produce just a single μop, while *complex* x86 instructions produce multiple μops. The potential for performance

degradation in this part of the processor is significant. In this paper, we highlight the performance degradation due to each component of this complex front end.

In Section 2, we show some of the previous work in this area. In Section 3, we present TAXI, our novel x86 performance infrastructure. The eight target applications are described in Section 4. Experimental results are presented and discussed in Section 5, and we conclude with Section 6.

2.0 Related Work

Lee et al. [7] provide some insight into Win32 applications by discussing some of their results with Etch, a general purpose tool for rewriting arbitrary Win32 binaries on x86 platforms without requiring modification of the source code. Their study compares some popular desktop applications to some SPECINT95 benchmarks in terms of application characteristics, cache behavior, TLB behavior, and branch prediction accuracy. Even though Etch is limited to user level traces only, some important findings about these applications are outlined. This work and [8] point to the difference between Win32 environments and the SPEC benchmark suite. Another study by Vlaovic and Uhlig [14] characterizes the event performance of some commercial "Natural I/O" applications (handwriting recognition, speech recognition, etc.) that run in the Win32 environment and compares them to the SPEC95 benchmarks. More recently, [13] reveal some interesting insights into five Win32 applications and their impact on branch target buffer performance, but again were limited to high level architectural events.

There have been many other processor performance evaluation studies. As long as we have had processors, there have been attempts at gathering information about the performance of applications that run on those processors. Kumar and Davidson[5] found that the major performance bottlenecks in the IBM 360/91 were the memory unit and the fixed point unit. Peuto and Shustek[9] apportioned the time both the IBM 370/168 Model 1 and the Amdahl 470 V6 spent among the various system components such as the cache, the instruction pipeline, and the individual instructions. Emer and Clark[3] characterized the VAX 11/780 by breaking into components related to particular op codes, operand specifiers, system components, and system events. More recently, there has been much work on characterizing different types of applications (e.g. java [4,10,11]), on various processors.

3.0 TAXI

To implement an out-of-order x86 simulator, we made use of two existing software tools, Bochs[6] and SimpleScalar[1]. In the following section, we introduce our simulation infrastructure, called **T**race **A**nalysis for **X**86 **I**nterpretation, or TAXI. The primary components of TAXI are made up of two parts: the functional simulation (Bochs portion), and the cycle-accurate simulation (SimpleScalar portion).

3.1 Bochs

Rather than starting from scratch, we used Bochs, an open-source Pentium emulator available from MadrakeSoft. 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 some few hundred contributors. Since the purchase of

Bochs by MandrakeSoft, it has been committed to Open-Source licensing (LGPL).

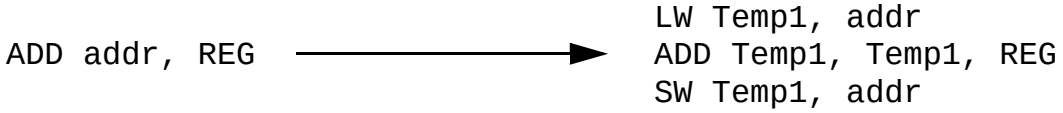
The advantage of using Bochs is that much of the hardware modeling is already completed. Bochs provides emulation for devices such as a keyboard, mouse, hard drive, floppy drive, and a VGA compatible monitor. In fact, it models not just the CPU, but the entire platform in enough detail to support the execution (with some debugging) 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 Virtual PC. This portion of TAXI, called the functional simulator, runs as a user-level process on a standard PC, modeling the platform components completely in software.

3.2 x86 Out-of-Order Simulator

Once again, since there already are a few available out-of-order simulators, we chose to use an existing software package rather than start from scratch. The package that has the most flexibility and provides open source code, is a simulator called SimpleScalar developed by Austin and Burger.

SimpleScalar performs fast, flexible, and reasonably accurate simulations of modern processors that implement the SimpleScalar architecture. Its native instruction set, called Pisa, is a derivative of the MIPS instruction set, and is relatively easy to extend and modify. The performance simulator portion of SimpleScalar has been aggressively tuned for speed, making it easier to run larger, more realistic applications.

Although SimpleScalar is a good starting point, it cannot directly handle x86 applications. 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 not unlike Pisa instructions. For instance, a memory-register add can be decomposed into the following Pisa instructions:



Other x86 instructions are decomposed similarly into Pisa instructions. We provide a breakdown for all addressing modes and for roughly 150 x86 instructions, which are sufficient for all of our benchmarks.

In addition to creating the μop breakdown, significant changes had to be made to the front end of the SimpleScalar pipeline as sketched in Figure 1. The output from Bochs is fed into the fetch stage of the simulator (SimpleScalar). Since x86 instructions are unaligned, a lot of work goes into determining instruction boundaries and bandwidth utilization.

The combination of the modified Bochs and augmented SimpleScalar are what comprise the heart of TAXI. However, if one only needs to do simple cache or branch prediction studies, then running the out-of-order simulator might be overkill. For this reason, the TAXI infrastructure includes other analysis tools.

For simple cache studies, dinero-IV[2] by Edler and Hill was modified to accept our trace files. For the instruction cache, the instruction trace is used as an input, with the pertinent information extracted, namely the effective instruction pointer (EIP) and the instruction length.

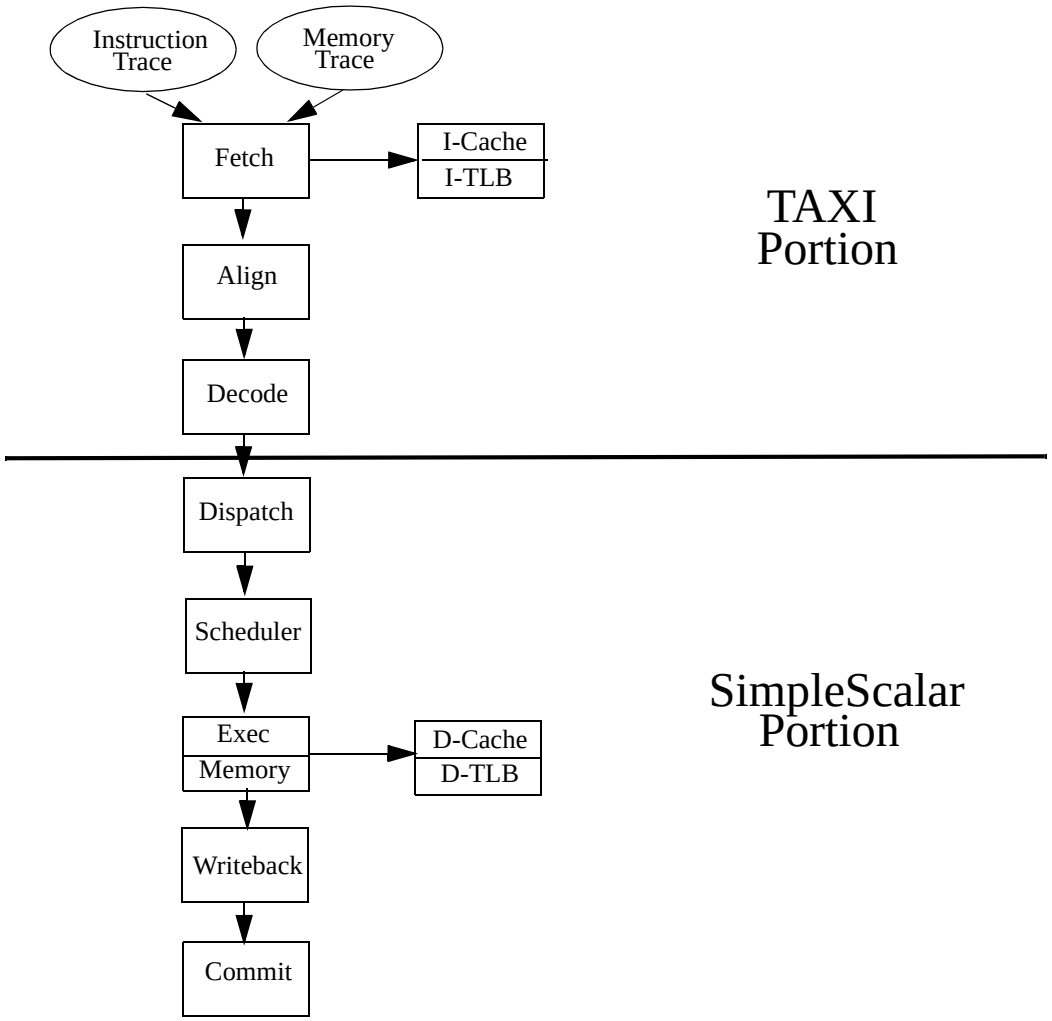


FIGURE 1. Functional pipeline for x86-outorder, TAXI portion is the front end of the pipeline, SimpleScalar is the back end

For the data cache, the address can be selected to be either the virtual address or the physical address (we ran the experiments with a 32M memory) with the loads corresponding to reads, stores corresponding to writes.

Another performance aspect that is useful to model without the out-of-order overhead is branch prediction. To do our branch prediction studies, the branch prediction mechanism of SimpleScalar was modified to accept an x86 instruction stream. For some of our branch target buffer (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 two simplified, specialized tools.

4.0 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. In the

Table 1: Application Trace Characteristics

	Insts. ($x10^6$)	**Uops/Inst**	**Insts/ Stores**	**Insts/ Loads**	**Inst/ Branch**
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
RealPlayer	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	4.81	2.68	5.09

Instructions per Branch column, `LOOP` and `REP` instructions are considered branches.

5.0 Experiments

The goal of this work is two-fold: first, we want to see the impact of non-aligned, dynamically interpreted, memory-to-register type instructions on the front-end of a given processor, and second, to attribute run time appropriately. In another words, we want to characterize how the behavior of the x86 instruction set affects the front-end of a processor that is similar to the front end of an Intel Pentium III. In this work, the configuration used is shown in Table 2 below. The number

Table 2: Baseline Configuration Parameters

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
Functional Units	4 IntALU, 1 IntMult/Div, 4 FP ALU, 1 FPMult/Div, 2 MemPorts
ROB	40 µop entries
Branch Predictor	2 level, 512 entry
BTB	512 entry 4-way
Return Address Stack	8 entries
Mispredict penalty	variable (no wrongpath execution)
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	18 cycle latency

of architected registers is the standard x86 register definition (8 general purpose registers and 8 floating point stack registers, plus control registers), but the number of physical registers is 64 integer and 64 floating point. The extra registers are used by the µops as temporary storage for memory operands, control state etc.

The pipeline in these studies is an eleven stage instruction pipeline, similar to that of the Pentium III, Pentium II, and Pentium Pro, all called the P6 architecture by Intel. A diagram of the pipeline is shown in Figure 2 above. The first 3 stages involve instruction fetch, while the next two stages are x86 instruction decode stages. The RAT stage is called the Register Alias Table stage, and this stage finds the location of operand dependencies. The ROB stage is where the processor allocates up to 3 µops to the ReOrder Buffer and/or passes them to the reservation station. The next two stages are dispatch and execute, respectively, while the last two stages are retirement and commit to x86 state. To assess the impact of this complex front end on the performance we begin with a configuration, called *opt*, that has an idealized (“optimal”) front end in which there is a perfect branch predictor, I-cache, and ITLB thereby keeping the Decoded Instruction Queue (DIQ) full. Then we successively replace each idealized component with the correspond-

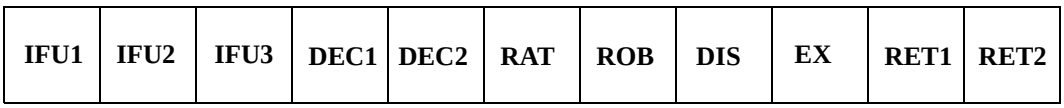


FIGURE 2. Pentium III Pipeline Structure

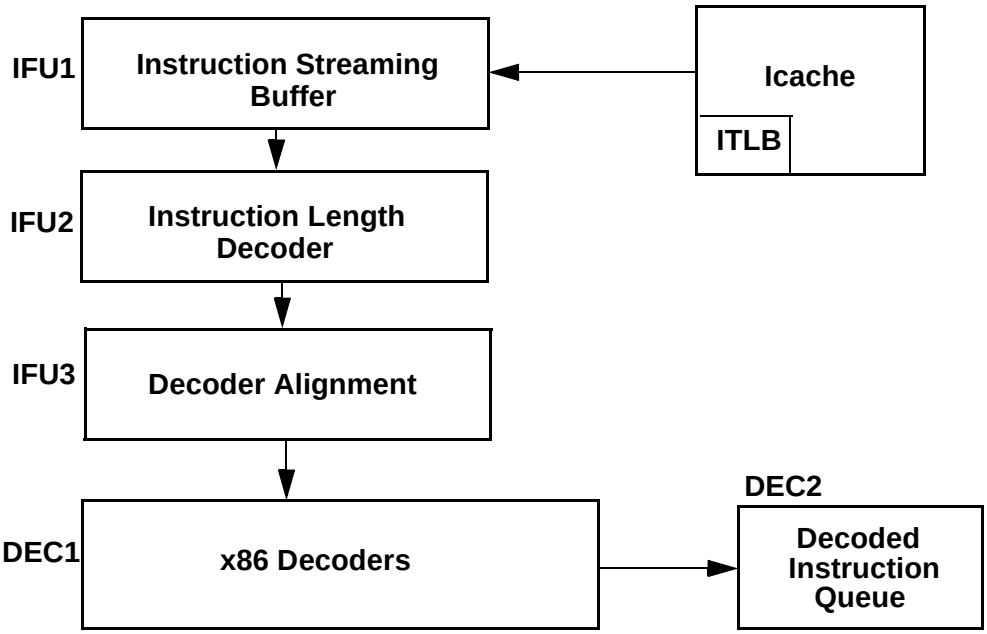


FIGURE 3. P6 Front-end Processor Block Diagram

ing component in the Baseline (P6) configuration as detailed in Table 2, and assess the performance penalty of each replacement. A block diagram of the P6 front end is shown in Figure 3.

The Baseline configuration has three decoders (two simple, one complex). Since our chosen methodology is to work our way back from DIQ, we first restrict the number of complex decoders to one, have unlimited simple decoders, and leave the remainder of the front end optimal. The performance penalty (increase in cycles per µop) caused by this restriction is called *cmplx dec*. Next, we restrict the number of decoders to two simple decoders and one complex decoder as in the Baseline configuration, and once again leave the remainder of the front end optimal. The resulting additional performance penalty is called *decoder.* Next, we restrict the decode width to 16 bytes (the *width* penalty). The final configurations successively introduce the Baseline streaming buffer from Table 2 (*SB* penalty), the branch predictor (*Bpred* penalty), the pipeline (increased from 8 back to 11 cycles, *pipe* penalty) and finally the Baseline I-cache (*Icache* penalty). This last configuration is in fact the Baseline configuration, as it has all the restrictions.

Figure 4 shows the cycles per µop (CPµ) for the Baseline configuration and the optimal configuration with a breakdown that shows how the difference in cycles attributed to each of the various restrictions in the Baseline system. We chose CPµ instead of Cycles per Instruction (CPI)

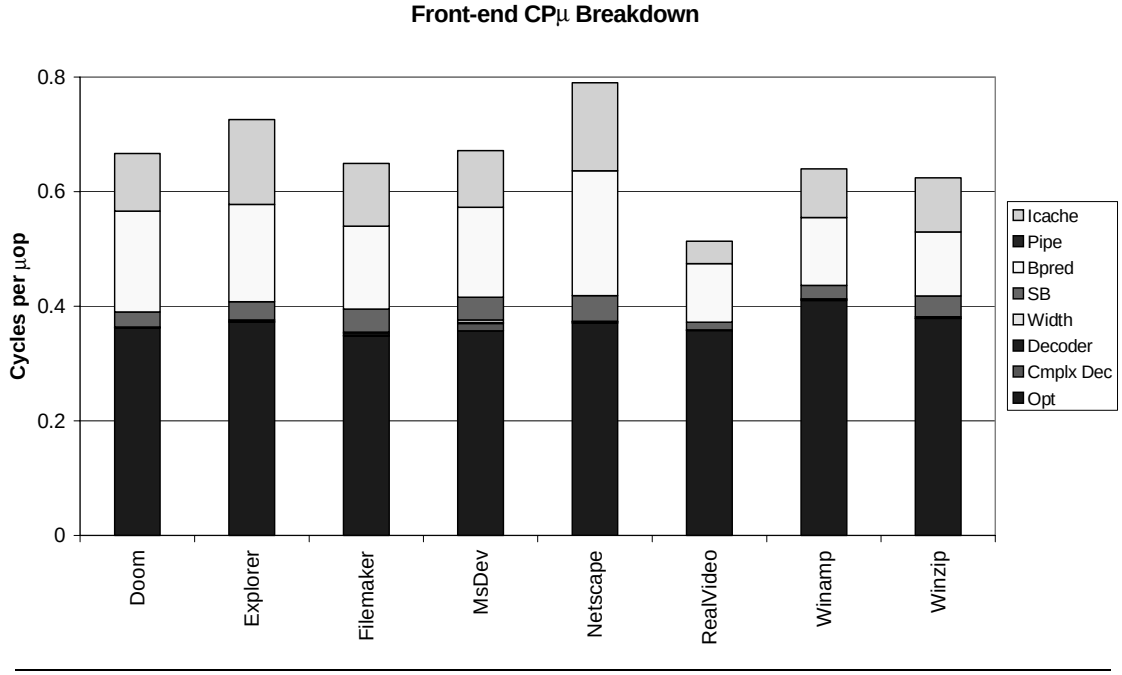


FIGURE 4. Front-end Cycles per μop breakdown for 8 Win32 Applications

since the amount of work per x86 instruction varies widely, whereas μops are more uniform. To get average CPI, one can multiply CPμ by μops/instruction from Table 1.

Our Win32 applications have an average CPμ of around 0.37. With the optimal front end, the main limiting factor is the size of the DIQ and the ROB, both of which are at near capacity during the runs of all the applications. With a maximum commit rate of 4 μops per cycle, an average CPμ of 0.37 translates into the retirement of 2.7 μops per cycle, making the back end realize only 67.5% of its maximum bandwidth. The average overall CPμ for the Baseline front end machine is 0.66 (1.5 μops per cycle), which is only 56% of the performance of the optimal front end.

Although the seven categories do contribute to this performance degradation, the overwhelming factors are the branch predictor and the I-cache. An average of 0.149 CPμ, (51% of the overall Baseline front end penalty, as seen in Figure 5) is added to each application due to imperfect branch prediction, and another 0.103 CPμ (36%) is due to the finite I-cache. One might have suspected that either the decoders, or perhaps the decode width or Streaming Buffer size might pose a serious bottleneck, but looking at Figure 4 this is obviously not the case.

As seen in the first column of Table 3, the average number of instructions that reside in the streaming buffer is around 1.78, and clearly 1 cache line in the streaming buffer is sufficient, as is a 16-byte decode width and 3 decoders. The next column in Table 3 is the average occupancy of the Decoded Instruction Queue, the third column contains the average number of entries in the

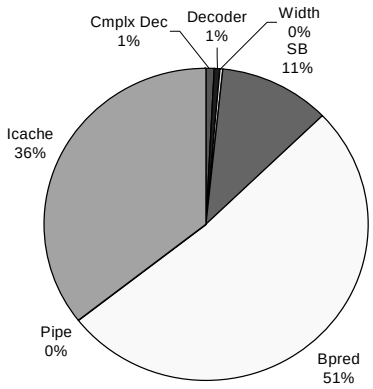


FIGURE 5. Percent of Lost Front end Cycles due to Each Restriction

Load/Store Queue and the fourth is the average occupancy of the Reorder Buffer. Our baseline

Table 3: Pentium III Baseline Results

	Avg. SB Occ. (insts)	Average DIQ Occ. (μops)	Avg. LSQ Occ. (μops)	Avg. ROB Occ. (μops)	L1 I-cache Misses ($x10^6$)	L1 D-cache Misses ($x10^6$)	L2 Cache Misses ($x10^6$)
Doom	1.78	1.63	3.82	8.75	13.0	2.98	0.46
Explorer	1.70	1.66	4.29	9.80	13.7	6.27	2.19
FileMaker	1.94	1.61	3.47	8.59	11.3	3.87	1.46
MsDev	1.78	1.77	4.54	9.96	8.83	9.54	2.25
Netscape	1.48	1.41	3.41	8.14	16.1	7.15	1.60
RealPlayer	2.40	2.38	6.50	16.0	5.97	11.4	0.30
Winamp	2.17	1.06	6.49	14.1	9.38	5.73	1.17
Winzip	2.04	2.23	4.68	13.8	5.44	6.23	1.59
Average	1.78	1.72	4.65	11.1	10.5	6.65	1.38

DIQ configuration has a DIQ size of 8 μops, which seems to be sufficient since its average occupancy is only 1.72 μops. However, caution must be exercised when forming conclusions based on averages. The occupancy figures are slightly pessimistic; since we do not currently execute down the wrong path, and merely stall the front end on a misprediction; the averages in the first four columns represent only μops on the correct execution path. The exact misprediction mechanism is described further in Section 5.1. Column five and seven in Table 3 show the number of misses in the L1 I-cache and the unified L2 cache which form the 36% performance loss due to the finite I-cache (see Figure 5).

5.1 Better Branch Prediction

Since the previous experiments highlighted the degradation caused by imperfect branch prediction, we chose to explore branch prediction further. The initial branch predictor configuration is a two-level 512 entry, with a 512 entry branch target buffer (BTB) and an 8 entry return address stack (RAS). The results for this predictor are shown in Table 3 below and are consistent with [13].

These results highlight why the branch predictor may have difficulty keeping the Streaming Buffer full. The *correct direction* column indicates the percentage of time that the predictor produces the correct direction; likewise, the *correct address* and *correct indirects* columns correspond to percentage of time that the Baseline predictor provides the correct address and correct indirect address, respectively. Since TAXI doesn't perform wrong path execution (because the Bochs traces do not include wrong path information), the actual branch penalty is difficult for TAXI to compute. If there is a misprediction, TAXI holds up the IFU1 stage until the branch that caused the misprediction is executed. Once the branch's address is computed, the processor can start fetching again. Although this method does not capture second-order effects and is a bit optimistic, it is reasonable measure. To compute the branch penalty, we ran the Baseline experiments again, but with perfect prediction and then subtracted the cycles of this run from the Baseline with the two-level 512 entry predictor. This difference, divided by the total number of branch mispredictions yields the average branch misprediction penalty and is shown in column four of Table 4. The 6.13 cycle resulting average penalty is smaller than what is commonly believed for the P6, but the effects of interrupts and exceptions are optimistically handled here, and other performance degradations that we have separated out may be lumped by others into the branch mispredict pen-

Table 4: Two-level, 512-entry Branch Predictor (Percent)

	Correct Direction	Correct Address	Correct Indirects	Avg. Mispredict Penalty (cycles)
Doom	76.7	61.1	34.1	6.77
Explorer	74.9	62.7	26.4	5.40
FileMaker	82.5	71.2	44.0	6.78
MsDev	81.0	72.5	31.4	5.22
Netscape	73.5	56.8	30.4	6.03
RealPlayer	90.6	83.9	50.6	7.73
Winamp	77.5	66.8	23.8	5.89
Winzip	80.7	72.7	29.0	5.24
Average	79.7	68.5	33.7	6.13

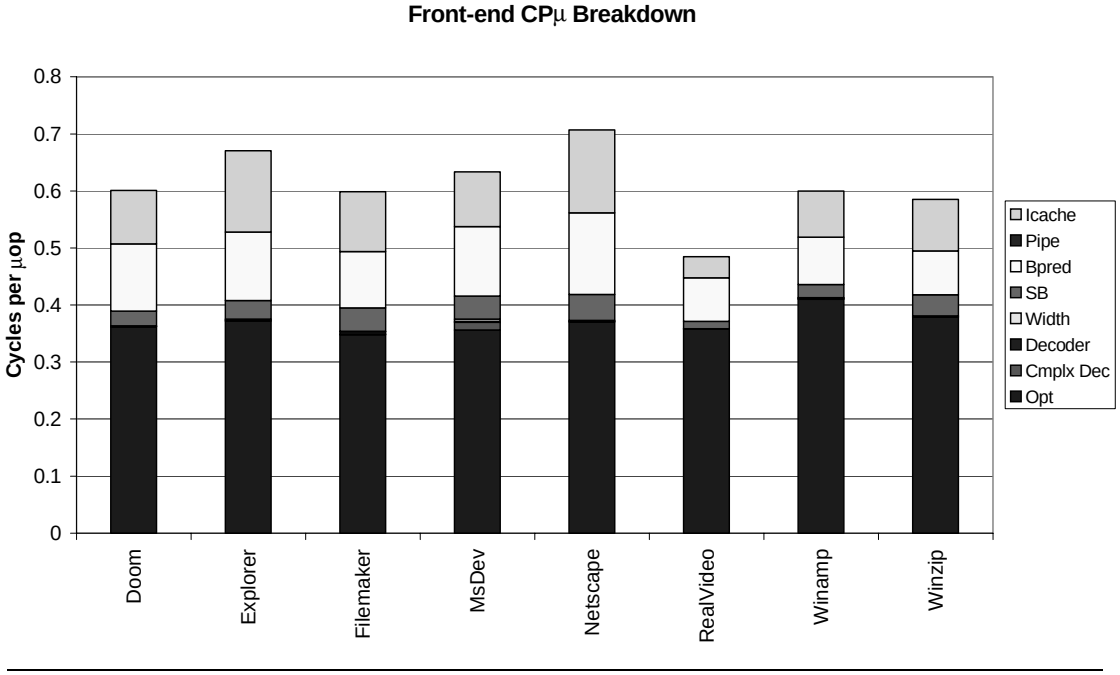


FIGURE 6. Front-end Cycles per μop breakdown for 8 Win32 Applications with a Combinational 16 K-entry GAg and 4 K-entry BTB

alty. Since the Baseline predictor comes up with the correct address nearly the 70% of the time, 30% of the branches in the trace see the 6.13 cycle branch penalty.

To assess branch prediction further, we changed the predictor in the Baseline system to a 16 K-entry GAg, a 4 K-entry BTB, and a 32 entry RAS. The performance improvement is shown in Figure 6. The average improvement over the Baseline scheme in table 2 is 7% less overall exe-

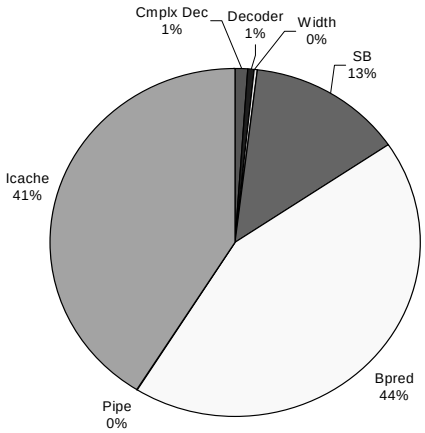


FIGURE 7. Percent of Lost Front end Cycles due to Each Restriction with GAg predictor

cution time. With this new predictor, the processor utilization characteristics are shown in Table 5.

Table 5: Pentium III GAg Predictor Results

	Avg. SB Occ. (insts)	Average DIQ Occ. (μops)	Avg. LSQ Occ. (μops)	Avg. ROB Occ. (μops)	Correct Direction (%)	Correct Address (%)	Correct Indirect (%)
Doom	2.00	1.83	4.41	10.1	88.6	79.9	47.5
Explorer	1.87	1.83	4.80	11.0	84.9	77.1	35.9
FileMaker	2.13	1.75	3.88	9.58	91.5	84.6	53.9
MsDev	1.91	1.89	4.92	10.8	87.6	81.3	39.0
Netscape	1.70	1.62	4.01	9.54	86.3	76.6	44.3
RealPlayer	2.56	2.54	7.05	17.3	95.2	90.7	61.7
Winamp	2.34	2.22	7.05	15.3	87.1	81.6	35.6
Winzip	2.20	2.41	5.16	15.1	88.7	83.9	38.5
Average	2.09	2.01	5.16	12.3	88.7	82.0	44.6

In Figure 7, the contribution to overall CPu by the branch predictor falls to 44%, while the other categories increase by a small percentage. In particular, the I-cache now comprises 41% of the perfromance penalty for the front end.

5.2 Better Predictor and Instruction Cache

After branch prediction, the next target is to improve the instruction cache. In this section, in addition to the improved branch predictor, the size of the I-cache was increased from 16

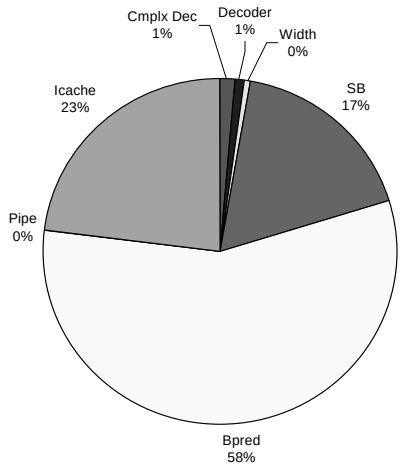


FIGURE 9. Percent of Lost Front end Cycles due to Each Restriction with GAg predictor and 64 KB I-cache

KBytes to 64 KBytes, maintaining 4-way associativity. This modification increased performance by 17% over the original Baseline system in Table 2.

In Figure 8, the CPμ category breakdown for each application is shown left to right, respectively, for the baseline configuration, the GAg predictor configuration, and finally, the GAg predictor with an increased I-cache. The pie chart in Figure 9 shows the new contributions to CPμ across all categories. Branch prediction once again becomes the number one degradation factor in

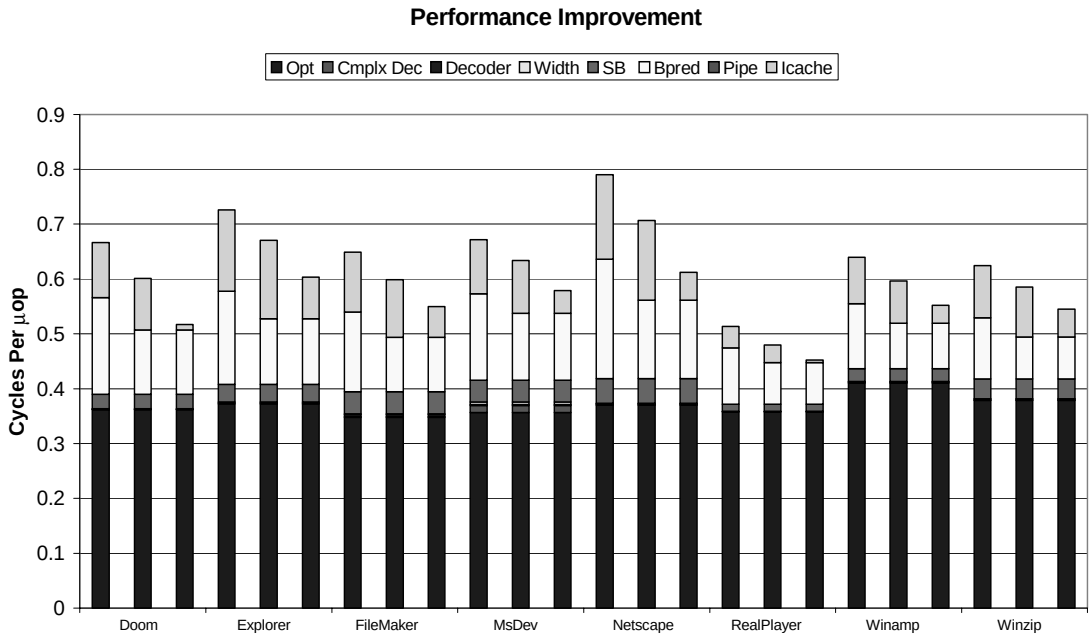


FIGURE 8. Front-end Cycles per μop breakdown for 8 Win32 Applications Comparison

front-end performance. The branch predictor now causes 58% of the front-end performance degradation. With the reduction in overall CPμ caused by both the improved branch predictor and the larger I-cache, the Streaming Buffer has jumped from 11% to 17% contribution to front-end performance loss.

6.0 Conclusions

We have introduced a new software tool for obtaining performance information for x86 processors running real workloads with real applications. With this simulation environment, called TAXI, we have been able to determine how the front-end performance bottleneck of a Pentium III-like processor is distributed. Although it has been suggested that this bottleneck might be primarily due to the decoders, the decode width, or even the streaming buffer, we have found that the vast majority of the penalty can be attributed to the branch prediction unit and the I-cache. By only targeting the branch prediction mechanism and doubling the size of the instruction cache, we were able to reduce overall run time in the eight Win32 applications by 17%.

The x86 instruction set architecture is much different than more RISC ISAs . Its variable length instructions, numerous addressing modes, and restricted architecture state make the implementation much more complex than for other processors, especially where high performance is important. With the information provided by TAXI, we were able to determine the number of cycles that each front end component has on overall performance and focus processor improvement efforts on those components that contribute the most to CPμ. We look forward to exploiting this new TAXI infrastructure to carry out further studies on x86 architectures, applications, and innovations.

7.0 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] J. S. Emer and D. W. Clark, “A Characterization of Processor Performance in the VAX-11/780,” *Proceedings of the Eleventh International Symposium on Computer Architecture*, pp. 310-310, June 1984.

[4] C.A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu, “A study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms,” *Proceedings of COMPCON*, pp.211-216, February 1997.

[5] B. Kumar and E. S. Davidson, “Computer system design using a hierarchal approach to performance evaluation,” *Communications ACM,* vol. 23, pp. 511--521, Sept. 1980

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

[7] 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.

[8] 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.

[9] B. L. Peuto and L. Shustek, “An Instruction Timing Model of CPU Performance,” *Proceedings of the International Symposium on Computer Architecture*, pp. 165-178, March 1977.

[10] R. Radhakrishnan, D. Talla, L. John, “Allowing for ILP in an Embedded Java Processor,” *Proceedings of the International Symposium on Computer Architecture*, pp. 294-305, June 2000, Vancouver, Canada

[11] R. Radhakrishnan, N. Vijaykrishnan, L. K. John and A. Sivasubramaniam, “Architectural Issues in Java Runtime Systems” *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 387-398, Toulouse, France, January 2000

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

[13] S. Vlaovic, E. S. Davidson, G. S. Tyson, “Improving BTB Performance in the Presence of DLLs,” *Proceedings of the 33rd Annual International Symposium on Microarchitecture*. October 1999.

[14] S. Vlaovic, R. Uhlig. “Performance of Natural I/O Applications,” *2nd Workshop on Workload Characterization*. October 1999.