Proc. CMG98, The 24'th Intl. Conf. for the Resource Mgt. and Perf. Eval. of Enterprise Computing Systems, December 6-11, 1998, Annaheim, CA. Vol. 1

Optimizing the performance of Wintel applications

Mark B. Friedman

Demand Technology Software

The best medicine for most sorts of performance problems is invariably preventative. Despite advance in software performance engineering [1,2], *developing complex computer programs that are both functionally correct and efficient remains a difficult and time-consuming task. This paper specifically looks at tuning Windows NT applications running on Intel hardware from the perspective of optimizing processor cycles and resource usage. Fine-tuning the execution path of code remains one of the fundamental disciplines of performance engineering.*

To bring this topic into focus, I will describe a case study where an application designed and developed specifically for the Microsoft Windows NT environment is subjected to a rigorous analysis of its performance using several commercially available CPU execution profiling tools. Since one of the development tools used to optimize the application program under consideration requires an understanding of the internal workings of Intel processors, this will justify an excursion into the area of Intel processor hardware performance.

An application tuning case study. The application that is the target of this analysis is a C language program that was written to collect Windows NT performance data continuously on an interval basis. The performance of this application is quite important. Since the app is designed primarily for use as a performance tool, it is very important that it run efficiently. A tool designed to diagnose performance problems should not itself be responsible for *causing* performance problems. Moreover, the customers for this application, many of whom are experienced Windows NT performance analysts, are a very demanding group of users.

Figure 1 illustrates the application structure and flow, which is straightforward. Following initialization, the program we developed enters a continuous data collection loop. Inside this loop, Windows NT services are called to retrieve selected performance data across a well-documented Win32 interface. The program, which consisted of a single executable module called `dmperfss.exe`, simply gathers performance statistics across this interface and logs the information collected to a local disk file. It is the optimization of the code within this inner loop that was the focus of this study.

Some additional details about the dmperfss application's structure and logic will be important to this discussion. To a large extent, the program's design is constrained by the Windows NT Win32 Application Programming Interface (API) that is the source of the performance data being collected. The performance data in Windows NT is structured as set of *Objects*, each with an associated set of *Counters*. (Individuals not accustomed to object-oriented programming terminology might feel more comfortable thinking about Objects as either *records* or rows of a database table and Counters as *fields* or columns in a database table.) There are approximately 100 different performance Objects defined in Windows NT at this writing, *base* Objects which are available on every system and *extended* Objects which are only available if specific application packages like MS SQL Server or Lotus Notes are installed. Within each Object, specific performance Counters are defined. Approximately nineteen different types of Counters exist, but

```
Initiation
loop until cycle end = TRUE;
    Win32 API calls to retrieve performance data;
    calculate;
    Write data to file;
end loop;
```

Figure 1.

generally NT Counters fall into three basic categories: accumulators, instantaneous measures, and compound variables. These categories of Counters are described in greater detail below.

Counter types. *Accumulators* are simple event Counters maintained within NT and various NT applications. A good example is a Counter which is incremented every time a page fault occurs. At the end of any given collection interval, this Counter records the accumulated total number of page faults that have occurred on the system in question since the time it was booted. A performance monitor application like `dmperfss.exe` gathers this information, subtracts the value obtained during the previous collections interval, and divides by the interval duration to derive a value that corresponds to the page fault rate during the last collection interval. *Instantaneous* Counters simply record the value of some metric right now. A good example is a Counter that reports the current number of Available Bytes. This is a single observation of this value at that moment in time, with no attempt to understand its range and variation over an interval.

The dmperfss application simply reports this current value; no additional calculations are involved. *Compound* variables are calculated measures where the performance monitoring interface provides all the values that needed to perform some predefined calculation. A good example is a Counter which reports the fraction of used disk space on an NT logical partition. The performance monitoring interface provides both the number of bytes used and capacity of the disk partition in a single compound Counter. A performance monitoring application like dmperfss must understand how to calculate the corresponding Counter value each interval.

The `dmperfss.exe` application retrieves designated Objects one at a time by making a call to the Win32 function `RegQueryEx`. Having collected a data sample, the program then makes the appropriate calculations for all the Counter variables of interest, and writes the corresponding Counter values to a data collection file. In some cases, there are multiple *instances* of an Object that need to be reported. For example, a computer system with four processors reports four instances of the processor Object each collection interval. In two cases, *parent-child* relationships are defined for Object instances. Logical disk instances have a parent Physical Disk instance, and threads are associated with a parent process. The significance of this relationship is that both the parent and child Object instances are retrieved using a single call to the performance monitoring interface.

At the time of the study, there were no apparent performance problems with the `dmperfss` data collection application. The program is most often used to retrieve a carefully crafted subset of the available data using collection intervals ranging from 1 to 15 minutes, with 5 five minutes being the average. At hose rates of data collection, the overhead of `dmperfss` data collection was uniformly much less than 1% additional processor utilization during the interval. Nevertheless, a good overall assessment of the performance of an application is almost always valuable in guiding future development. Furthermore, there are good reasons to run data collection at much more frequent intervals than current customer practice.[1] Consequently, we wished to investigate whether it would be feasible to build a monitoring program that would collect certain data at much more frequent intervals – perhaps as frequently as once per second, or even more frequently.

The development tools. The analysis of the performance of this application program was initiated at a point in the development cycle where the code was reasonably mature and stable. Once a program is functional, it is appropriate to tackle performance optimization. *It* should be stressed that performance considerations should be taken into account at every stage of application design, development, and deployment.

The study focused on analyzing the code execution path using commercially available profiling tools. CPU profiling tools evaluate a program while it is executing and report on which sections of code are executing as a proportion of overall execution time. The execution profile thus derived allows programmers to focus on particular routines that account for the most execution time delay. This sort of information supplies an empirical element to performance-oriented program design and development, which is otherwise sorely missing. Without reliable, quantitative information on code execution paths, programmers tend to rely on very subjective criteria to make decisions that affect application performance. I like to say that these tools eliminate a lot of idle chatter around the coffee machine about why a program is running slowly and what can be done to fix it. Profilers provide programmers with hard evidence of where their programs are spending their time during execution.

A code profiler also provides hard data to help evaluate alternative approaches that a programmer

[1] *For instance, accumulator values, which are running totals, and instantaneous values are both collected at the same rate. Would it be possible to collect instantaneous values more frequently and then summarize these sample observations? See [3] for a good discussion of the reasons for running data collection at short intervals.*

might consider to speed up program execution. For instance, which compiler optimization options are worthwhile, which sections of code should be targeted for revision, and where inline assembler routines might prove most helpful. In planning for the next cycle of development, the results of a code execution profile improve the decision-making process.

The code in the program under analysis here, as most programs written for the Windows environment do, makes extensive use of the Win32 application programming interface and C run time services. One highly desirable outcome of a code profiling exercise is greater understanding of the performance impact of various system services, which is where many programs are spending the majority of their time during execution. The time spent inside these calls to system services is a like black box, which means the programmer can do very little to influence their performance. But understanding their performance impact at least helps the programmer understand how to use them more effectively. In cases where there are no viable alternatives to these system services, the programmer can learn to interact with them more effectively.

The use of the following profiling tools was investigated during the course of this study:

- the profiler option in the Microsoft Visual C++ compiler version 5,
- the Rational Visual Quantify execution profiler, and
- the Intel vTune version 2.5 optimization tool.

All three are popular commercial packages. In selecting these and only these specific tools, no attempt was made to provide encyclopedic coverage of all the code profiling options that are available for the Windows NT/Intel environment. Instead, we tried to focus on an in-depth analysis using a few of the better known and widely available tools for NT program development to solve a real-world problem.

The Microsoft Visual C++ optimizing compiler was a natural choice because all the code development was performed using this tool. It is a widely used compiler for this environment, and its built-in code profiling tool is a natural first choice for developers of all kinds who might be reluctant to incur the expense of an additional software package. The Rational Visual Quantify program is one of the better known profiler tools for C and C++ language development. Rational is a leading manufacturer of developer tools for Unix, Windows, and Windows NT. The Visual Quantify program features integration with the Microsoft Visual Studio development environment. It is usually reviewed in surveys of C++ development tools published in popular trade publications. [4,5] Finally, the Intel vTune program was used because this has garnered wide acceptance within the development community as an optimization tool developed by Intel specifically for programs that run on Intel hardware. vTune is a standalone program, and, we discovered, is more oriented toward assembly language development than typical application development using C or C++.

Overall, we found both add-on packages to be extremely useful and well worth their modest investment. Visual Quantify provides a highly intuitive user interface. It extends and widens the CPU profiling information that is available through the built-in facilities of MS Visual C++ by incorporating information on many system services. We found it greatly increased our understanding of the code's interaction with various Windows NT system services. VTune supplemented this understanding with even more detailed information about our code's interaction with the NT run-time environment. VTune's unique analysis of Intel hardware performance provides singular insight into this critical area of application performance.

The Case Study.

Visual C++ profiler.

The first stage of the analysis used the code execution profiler built into MS Visual C++. This is a natural place for any tuning project to begin given that no additional software packages need to be licensed and installed. Run-time profiling is enabled from the **Link** Tab of the **Project**, **Settings** Dialog box inside Developer Studio, which also turns off incremental linking. Profiling can be performed at either the function or line level. Function level profiling provides the number of calls to the function and adds code which keeps track of the time spent in execution while resident in the function. Line profiling is much higher impact and also requires that debug code be generated. In this exercise, I will only report on the results of function profiling.

You select **Profile** from the **Tools** menu to initiate a profiling run. Additional profiling options are available at run time. The most important of these allow the user to collect statistics only on specific modules within a program or restrict line profiling to certain specific lines of code. At the conclusion of the run, a text report is available in the Output window under the **Profile** Tab. You can view the report there or import the text into a spreadsheet where the output text can be sorted and further manipulated. The first few lines of the function level Profile report are illustrated in Figure 2. The report output is automatically sorted in the most useful sequence, showing the functions in order by the amount of time spent in the function.

The data in Figure 2 is from a profiling run of the dmperfss program that lasted about six minutes. The program was set to collect a default collection set once a second and write this information to a file. The following performance Objects are included in this collection set: System, Processor, Memory, Cache, Logical Disk, Physical Disk, Process, Redirector, Server, and Network Segment.

The profiling report itself is largely self-explanatory. The amount time spent in the function, the amount time spent in the function and its descendants, and a Hit Count are tabulated. The specific function is identified by name, with the name of the object module that exports the function beside it in parentheses.

Interpreting the report is equally straightforward. The profiling statistics indicate that the program spent 87.5% of its execution time in a function called `WaitOnEvent`. The event the program is waiting for in this function is the Windows NT Timer event which signal that it is time to collect the next data sample. Notice that the Hit Count for this function, the number of times this function was entered, is just slightly less than the number of seconds in the execution interval. This no doubt reflects some delay in initialization. It seems safe to assume that the program is actually executing the other 12.5% of the time. The WriteDataToFile function, which was entered 393,329 times accounts for close to 25% of the program's actual execution time. It appears that this function is called over 1,000 times for each collection interval.

The only noticeable difficulty in working with the Visual C++ profile report is that it does not identify the function call parent-child relationships. Notice that the Function call `FindPreviousObjectInstanceCounter` is executing 1.2% of the time (about 10% of the program's actual active time). When functions called by `FindPreviousObjectInstanceCounter` are factored in, the function call is executing 4.5% of the time. Unfortunately, it is not possible to identify the child functions that are called from a parent function by glancing at the report. Since the programmer creating the execution profile also has access to the source code, it should be reasonably simple to trace the function's execution using the debugger, for example, to determine what calls the `FindPreviousObjectInstanceCounter` function makes to other helper functions. In practice, for a program as complex as dmperfss is, this is not a trivial exercise.

In practice, while the built-in execution profiler is not as robust as other available tools, it did help us identify the programs modules that were having the greatest impact on performance. The C++ Profile report was consistent with later analysis results that honed in on the same relatively small set of modules that were responsible for a disproportionate amount of CPU time consumption. While it was not as comprehensive as the other tools we compared it to, the Microsoft Visual C++ proved to be both easy to use and effective.

Visual Quantify.

Visual Quantify, available from Rational, is an add-on code profiling tool available for Windows NT that works with Microsoft Visual C++, Visual Basic, and Java running on Intel hardware. It addresses the major usability limitations we encountered with the profiler built into the Microsoft Compiler. First, and foremost, it has an easy-to-use and intuitive GUI interface that makes it easy to navigate through the profiling output.

```
  Module Statistics for dmperfss.exe
  ---------------------------------

  Time in module: 283541.261 millisecond
  Percent of time in module: 100.0%
  Functions in module: 155
  Hits in module: 11616795
  Module function coverage: 72.3%


      FUNC          FUNC+CHILD          HIT

      TIME    %        TIME       %     COUNT   FUNCTION
 ---------------------------------------------------------
248146.507  87.5  248146.507   87.5       249 _WaitOnEvenet (dmwrdata.obj)
  8795.822   3.1    8795.822    3.1    393329 _WriteDataToFile (dmwrdata.obj)
  4413.518   1.6    4413.518    1.6      2750 _GetPerfDataFromRegistry (dmwrdata.obj)
  3281.442   1.2    8153.656    2.9    170615 _FormatWriteThisObjectCounter (dmwrdata.obj)
  3268.991   1.2   12737.758    4.5     96912 _FindPreviousObjectInstanceCounter (dmwrdata.obj)
  2951.455   1.0    2951.455    1.0   3330628 _NextCounterDef (dmwrdata.obj)
```

Figure 2.

But, Visual Quantify (or VQ, for short), I found, is more than just a pretty face. The profiling data it collects extends to a surprising number of system modules and services. VQ measures the performance impact associated with third-party ActiveX and DLL-based components that your program links to dynamically. It also captures the amount of processor spent inside a surprising number of Win32 API calls. Acquiring profiling information on the overhead that the `dmperfss` program spent *outside* the executable certainly proved to be of great benefit in this particular project.

Like the MS VC++ profiler, VQ gathers information at the level of either the function or the line. VQ provides additional options which allow you to specify when you want the profiling data collection effort to begin. This lets you, for example, bypass initialization code that you do not care about so that you can focus on the application's inner loop. Another nice feature is the ability to compare two or more profiling runs of the same program. This is very useful during the iterative process of implementing code changes to improve performance to ensure that the changes are having the desired effect.

VQ obtains profiling information on dynamically loaded modules called from the target application at run-time. It runs the application being monitored in its address space and watches as it calls other modules. So long as the external function being called is associated with a sharable dll, VQ can instrument it. (Fortunately, most DLLs and OCXs *are* sharable.) It does this by copying the module being called into the VQ process address space and inserting its measurement hooks into this private copy of the code. Rational calls this technique Object Code Insertion, which it claims to \have patented.

The first noticeable thing about running Visual Quantify is how long it takes to get started. This is because the process of intercepting dynamic program loads, loading a private copy of the module into the VQ process address space, and inserting the instrumentation hooks is quite lengthy. Commonly accessed DLLs like MCVSRT40.dll are normally resident in the system when a program begins execution and do not have to be loaded. In the case of the `dmperfss` program, VQ initialization took over twenty minutes before program execution was able to proceed unimpeded. VQ keeps you entertained during this startup delay with two displays that show the progress of initialization. The Threads display, illustrated in Figure 3, appears by default showing the status of the profiling run. The Threads display is a histogram showing activity by thread. I am not certain why so many threads are displayed here – the application itself only has two, corresponding to main_66 (top) and thread_17f (the mainly blue

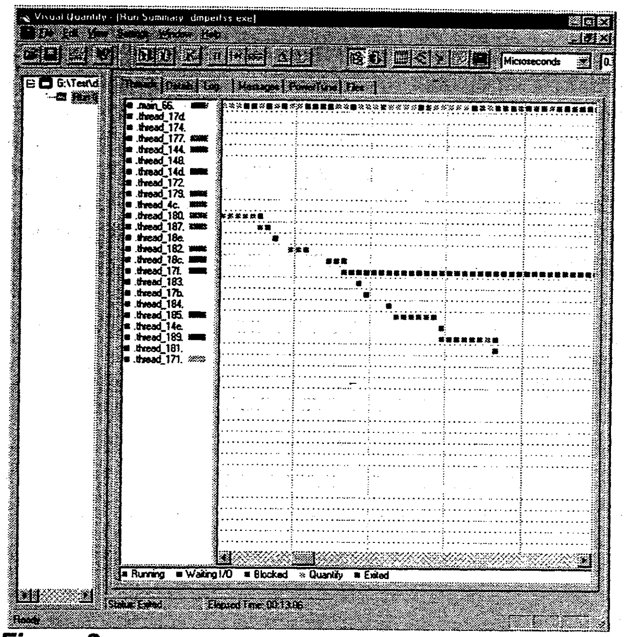
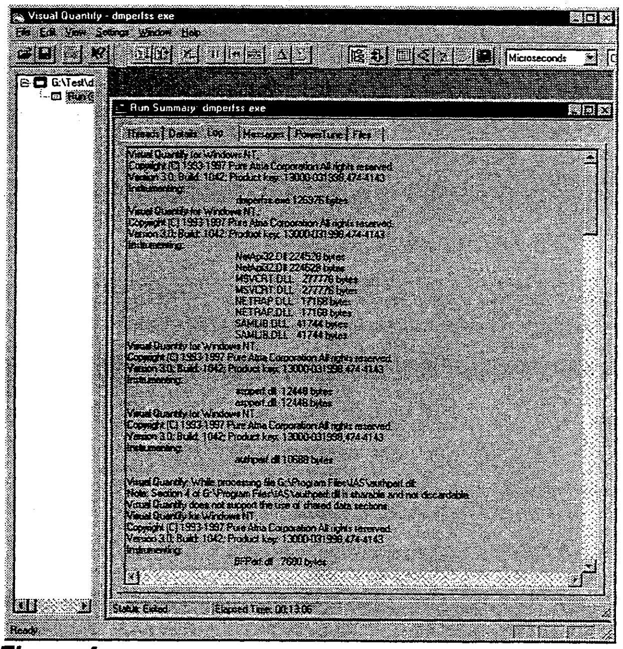


Figure 3.

squares in the middle of the screen). It seems plausible that VQ spawns a separate thread to load each dll. Gray squares in the main_66 thread indicate VQ delays to load the modules and insert its hooks.

The Log display illustrated in Figure 4 is quite informative. VQ logs each load module as it encounters it in the course of program execution. Here we see `dmperfss.exe` being instrumented, followed by `NetApi32.DLL`, `MSVCRT.DLL`, `NETRAP.DLL`, `SAMLIB.DLL`, etc. When VQ encounters a module that it cannot instrument, like `authperf.dll`, a warning message is


Visual Quantify - dmperfss exe
File Edit View Settings Window Help
Microseconds
G:\Test\d
Run Summary: dmperfss exe
Threads Details Log Messages PowerTune Files
Status: Exited Elapsed Time: 00:13:06
Ready

```
Visual Quantify for Windows NT.
Copyright (C) 1993-1997 Pure Atria Corporation All rights reserved.
Version 3.0; Build: 1042; Product key: 13000-031998.474-4143
Instrumenting:
                        dmperfss.exe 126976 bytes
Visual Quantify for Windows NT.
Copyright (C) 1993-1997 Pure Atria Corporation All rights reserved.
Version 3.0; Build: 1042; Product key: 13000-031998.474-4143
Instrumenting:
                        NetApi32.Dll 224528 bytes
                        NetApi32.Dll 224528 bytes
                        MSVCRT.DLL   277776 bytes
                        MSVCRT.DLL   277776 bytes
                        NETRAP.DLL   17168 bytes
                        NETRAP.DLL   17168 bytes
                        SAMLIB.DLL   41744 bytes
                        SAMLIB.DLL   41744 bytes
Visual Quantify for Windows NT.
Copyright (C) 1993-1997 Pure Atria Corporation All rights reserved.
Version 3.0; Build: 1042; Product key: 13000-031998.474-4143
Instrumenting:
                        aspperf.dll 12448 bytes
                        aspperf.dll 12448 bytes
Visual Quantify for Windows NT.
Copyright (C) 1993-1997 Pure Atria Corporation All rights reserved.
Version 3.0; Build: 1042; Product key: 13000-031998.474-4143
Instrumenting:
                        authperf.dll 10688 bytes

Visual Quantify: While processing file G:\Program Files\IAS\authperf.dll:
Note: Section 4 of G:\Program Files\IAS\authperf.dll is sharable and not discardable.
Visual Quantify does not support the use of shared data sections.
Visual Quantify for Windows NT.
Copyright (C) 1993-1997 Pure Atria Corporation All rights reserved.
Version 3.0; Build: 1042; Product key: 13000-031998.474-4143
Instrumenting:
                        BFPerf.dll  7680 bytes
```

Figure 4.

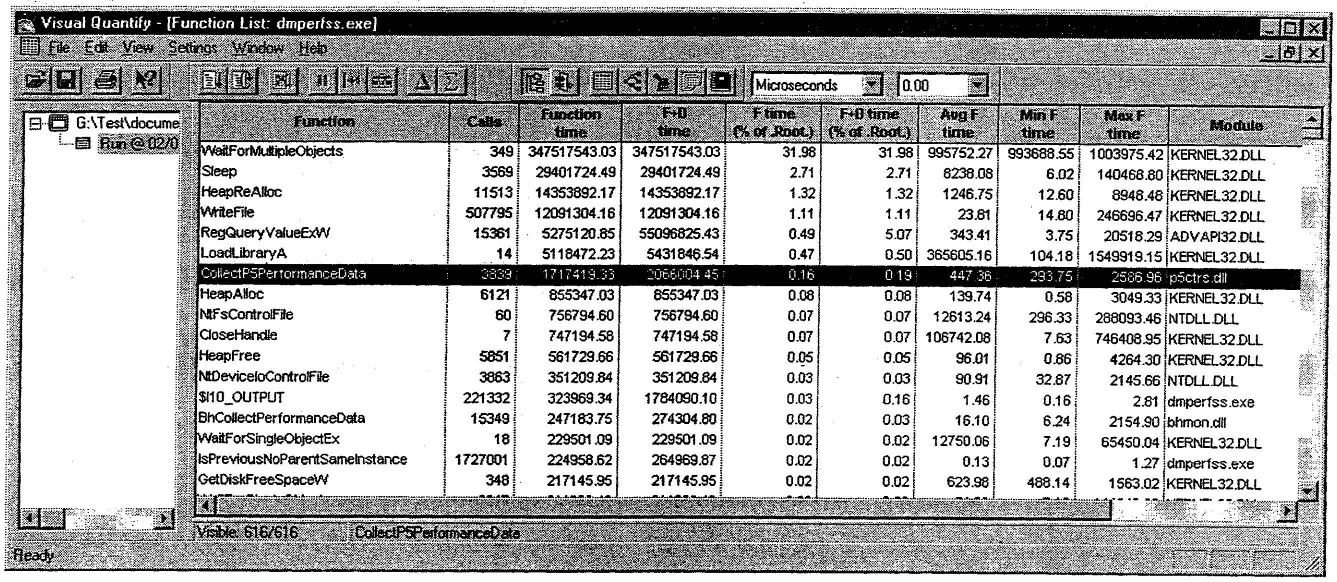
printed. Obviously, following the trail of the modules being called from the main application program, loading them, and instrumenting them is very time-consuming. One of VQ's execution time options copies instrumented load modules to a disk cache. On subsequent VQ runs, the program checks first to see if a current copy of the load module is available from the disk cache. This speeds up processing on subsequent profiling runs considerably.

Figure 5, which displays the `dmperfss` functions in order by the amount time spent in execution, shows just how effective VQ is in instrumenting third party code. Where the Microsoft VC++ profiler was able to collect data on 155 function calls, VQ provides data on 616. Eight out of the first eleven modules are functions embedded in KERNEL32.DLL. VQ shows precisely how long these Win32 API calls take. The first dmperfss function to make the list is $10_OUTPUT, which is responsible for only 0.03 % of the total execution time. The information VQ provides changed our perspective from fine tuning the code in `dmperfss` to improving the way the code interacts with Windows NT system services. Because it gathers all this additional information, VQ's intuitive user interface is even more important. You can open several different windows at one time, each focusing on a different segment of the program, and you navigate easily through the code execution path. The screen shots here cannot do justice to just how effectively designed the interface is.

Of course, this added level of information comes at a price in additional overhead during the profiling run. Notice that the function call WaitForMultipleObjects accounts for 32% of the execution time. This is the KERNEL32.DLL function, which is called from the `WaitOnEvent` routine located in `dmwrdata.obj` that the C++ profiler found was consuming fully 87.5% of the time! Because of the amount of measurement overhead, it is important to accept VQ's function timing statistics as relative indicators of performance, rather than absolute numbers. In other words, the proportion of the time the `dmperfss` program spent in `WaitOnEvent` compared to `WriteDataToFile` was consistent across the MS VC++ profiler and VQ. I will describe the effort I made to validate the measurements from all three sources at the conclusion of this article.

The amount of overhead associated with the VQ measurement technology was enough of a factor to limit the types of program execution scenarios that I could realistically gather performance information about. In particular, I wanted to assess the performance impact of collecting different NT performance Objects. The easiest way to do that was to set up a data collection set that included *all* available Objects, overriding the program's default collection set to use the Master Collection set instead. However, when I tried to run VQ with `dmperfss` collecting all available Objects, there was so much measurement overhead that I was unable to collect data at frequent enough intervals to gather data in a systematic fashion. I was able to work around the problem easily enough by creating subsets of the program's Master Collection set and analyzing them. The overhead associated with VQ profiling is the one consideration that you have to be aware of when you are using the program and interpreting its results.

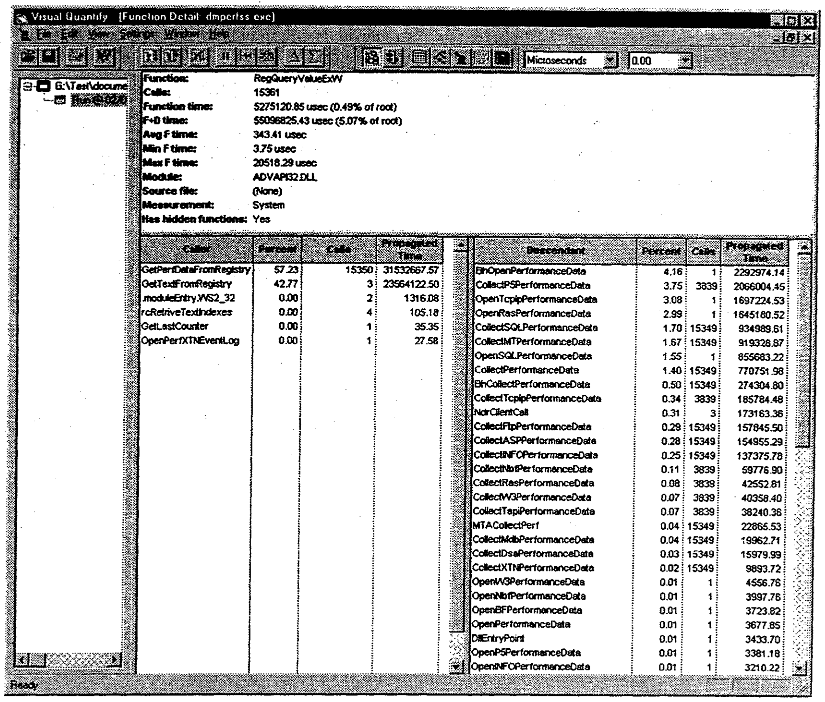
Where VQ excelled was in illuminating the execution path associated with the Win32 services the `dmperfss` program routinely called and which were responsible for much of the execution time. Figure 6 shows the Function Detail display that is obtained by zooming in on the `RegQueryValueEx` function, which is the interface



Function	Calls	Function time	F+D time	F time (% of Root)	F+D time (% of Root)	Avg F time	Min F time	Max F time	Module
WaitForMultipleObjects	349	347517543.03	347517543.03	31.98	31.98	995752.27	993688.55	1003975.42	KERNEL32.DLL
Sleep	3569	29401724.49	29401724.49	2.71	2.71	8238.08	6.02	140468.80	KERNEL32.DLL
HeapReAlloc	11513	14353892.17	14353892.17	1.32	1.32	1246.75	12.60	8948.48	KERNEL32.DLL
WriteFile	507795	12091304.16	12091304.16	1.11	1.11	23.81	14.80	246696.47	KERNEL32.DLL
RegQueryValueExW	15361	5275120.85	55096825.43	0.49	5.07	343.41	3.75	20518.29	ADVAPI32.DLL
LoadLibraryA	14	5118472.23	5431846.54	0.47	0.50	365605.16	104.18	1549919.15	KERNEL32.DLL
CollectP5PerformanceData	3839	1717419.33	2066004.45	0.16	0.19	447.36	293.75	2586.96	p5ctrs.dll
HeapAlloc	6121	855347.03	855347.03	0.08	0.08	139.74	0.58	3049.33	KERNEL32.DLL
NtFsControlFile	60	756794.60	756794.60	0.07	0.07	12613.24	296.33	288093.46	NTDLL.DLL
CloseHandle	7	747194.58	747194.58	0.07	0.07	106742.08	7.63	746408.95	KERNEL32.DLL
HeapFree	5851	561729.66	561729.66	0.05	0.05	96.01	0.86	4264.30	KERNEL32.DLL
NtDeviceIoControlFile	3863	351209.84	351209.84	0.03	0.03	90.91	32.87	2145.66	NTDLL.DLL
$I10_OUTPUT	221332	323969.34	1784090.10	0.03	0.16	1.46	0.16	2.81	dmperfss.exe
BhCollectPerformanceData	15349	247183.75	274304.80	0.02	0.03	16.10	6.24	2154.90	bhmon.dll
WaitForSingleObjectEx	18	229501.09	229501.09	0.02	0.02	12750.06	7.19	65450.04	KERNEL32.DLL
IsPreviousNoParentSameInstance	1727001	224958.62	264969.87	0.02	0.02	0.13	0.07	1.27	dmperfss.exe
GetDiskFreeSpaceW	348	217145.95	217145.95	0.02	0.02	623.98	488.14	1563.02	KERNEL32.DLL

Figure 5.

used to collect performance data in NT. The Function Detail screen displays the program functions that called RegQueryValueEx and the functions called from RegQueryValueEx. (You can navigate back and forth in the Function Detail to trace the execution path forward and back.) In the absence of the information VQ provides, the interface to RegQueryValueEx is a Black Box. The calls to RegQueryValueEx are well-documented in the Win32 System Development Kit (SDK). Microsoft also supplies a code sample in the SDK that shows how to support extended Counters across this interface. But the actual program flow of control is not documented. VQ opens up this Black Box so that we can get a good look inside.


Visual Quantify [Function Detail: dmperfss.exe]
File Edit View Settings Window Help
Microseconds
0.00
G:\Test\docume
Ready

Function:	RegQueryValueExW
Calls:	15361
Function time:	5275120.85 usec (0.49% of root)
F+D time:	55096825.43 usec (5.07% of root)
Avg F time:	343.41 usec
Min F time:	3.75 usec
Max F time:	20518.29 usec
Module:	ADVAPI32.DLL
Source file:	(None)
Measurement:	System
Has hidden functions:	Yes

Caller	Percent	Calls	Propagated Time
GetPerfDataFromRegistry	57.23	15350	31532667.57
GetTextFromRegistry	42.77	3	23564122.50
.moduleEntry.WS2_32	0.00	2	1316.08
rcRetriveTextIndexes	0.00	4	105.18
GetLastCounter	0.00	1	35.35
OpenPerfXTNEventLog	0.00	1	27.58

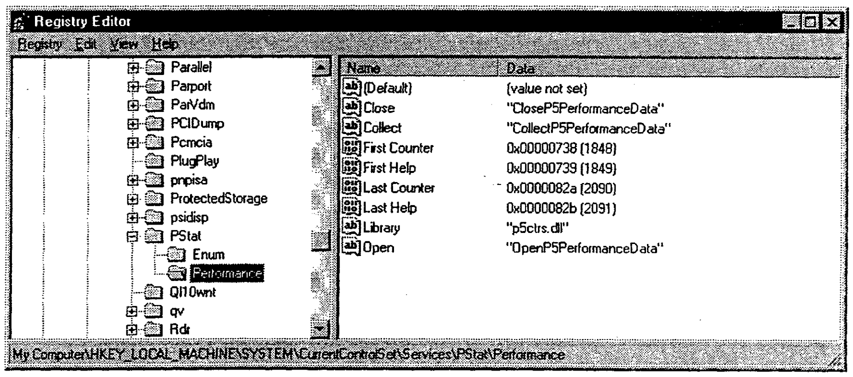
Descendant	Percent	Calls	Propagated Time
BhOpenPerformanceData	4.16	1	2292974.14
CollectP5PerformanceData	3.75	3839	2066004.45
OpenTcpipPerformanceData	3.08	1	1697224.53
OpenRasPerformanceData	2.99	1	1645180.52
CollectSQLPerformanceData	1.70	15349	934989.61
CollectMTPerformanceData	1.67	15349	919328.87
OpenSQLPerformanceData	1.55	1	855683.22
CollectPerformanceData	1.40	15349	770751.98
BhCollectPerformanceData	0.50	15349	274304.80
CollectTcpipPerformanceData	0.34	3839	185784.48
NdrClientCall	0.31	3	173163.36
CollectFtpPerformanceData	0.29	15349	157845.50
CollectASPPerformanceData	0.28	15349	154955.29
CollectINFOPerformanceData	0.25	15349	137375.78
CollectNbfPerformanceData	0.11	3839	59776.90
CollectRasPerformanceData	0.08	3839	42552.81
CollectW3PerformanceData	0.07	3839	40358.40
CollectTapiPerformanceData	0.07	3839	38240.36
MTACollectPerf	0.04	15349	22865.53
CollectMdbPerformanceData	0.04	15349	19962.71
CollectDsaPerformanceData	0.03	15349	15979.99
CollectXTNPerformanceData	0.02	15349	9893.72
OpenW3PerformanceData	0.01	1	4556.76
OpenNbfPerformanceData	0.01	1	3997.76
OpenBFPerformanceData	0.01	1	3723.82
OpenPerformanceData	0.01	1	3677.85
DllEntryPoint	0.01	1	3433.70
OpenP5PerformanceData	0.01	1	3381.18
OpenINFOPerformanceData	0.01	1	3210.22

Figure 6.

The NT performance monitor interface. Extended performance Objects in NT are associated with a perflib DLL which responds to three function calls: Open, Close, and Collect. Figure 7 illustrates the Registry entries that a perflib DLL must create. A performance monitor application like the NT Performance Monitor or dmperfss scans the NT Registry looking for these entries in order to discover what collection data is available on a particular machine. The Open function for the performance DLL is then called initially by the performance monitoring application to enumerate the Objects and Counters that it can report. The Collect function is then performed at regular intervals to retrieve data. The metadata retrieved from the Open call is used to process the 'ata buffers retrieved by the Collect function. The Close function is used to perform any necessary cleanup.

Figure 6 showed that RegQueryValueExW is called from two places by dmperfss: 15,350 times from GetPerfDataFromRegistry and just 3 times from GetTextFromRegistry. GetPerfDataFromRegistry is called from the dmperfss data collection loop once an interval for each performance Object, while GetTextFromRegistry is called only at initialization to retrieve the performance Object metadata. Yet the 3 initialization calls are almost as time-consuming as the interval data collection calls, according to the VQ measurement data.

VQ then breaks down the time spent in RegQueryValueEx according to the functions it calls, most of which are Open, Close, and Collect calls embedded in perllib DLLs. From the number of function calls executed, we see a one-to-one correspondence between calls to RegQueryValueEx and a number of different perflib DLL Collect functions. What we found surprising was that functions like CollectSQLPerformanceData, which is associated with the collection of MS SQL Server performance data, were being called implicitly. MS SQL Server was installed on this machine, but the dmperfss program did *not* explicitly reference the SQL Server performance Objects. As expected, the perflib DLL Open calls are made just once. Several of the Open calls are very time-consuming, possibly due to serialization delays associated with interprocess communication. But since the Open calls are made just once, the time spent waiting for the perflib DLL to initialize is acceptable. (Besides, there is nothing we can do to improve someone else's code.)

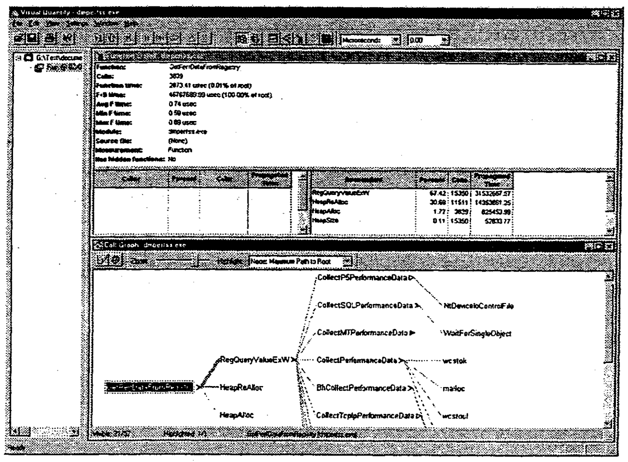


Name	Data
(Default)	(value not set)
Close	"CloseP5PerformanceData"
Collect	"CollectP5PerformanceData"
First Counter	0x00000738 (1848)
First Help	0x00000739 (1849)
Last Counter	0x0000082a (2090)
Last Help	0x0000082b (2091)
Library	"p5ctrs.dll"
Open	"OpenP5PerformanceData"

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PStat\Performance

Figure 7.

Figure 8 shows a graphical view of the execution profile that VQ creates called the Call Graph. The Call Graph traces the subfunctions called from function `GetPerfDataFromRegistry` in the `dmperfss.exe` program. The thickness of the line connecting two functions represents relative amount of time traversing that particular logical path. This display clearly identifies the program's critical path of execution, very useful for zeroing on what sections of code need work. Recall that using the Microsoft VC++ profiler, it was not possible to measure the execution time in any of the modules called from `GetPerfDataFromRegistry` because they were outside the range of the



```
Function:            GetPerfDataFromRegistry
Calls:               3839
Function time:       2873.41 usec (0.01% of root)
F+D time:            46767689.59 usec (100.00% of root)
Avg F time:          0.74 usec
Min F time:          0.59 usec
Max F time:          0.89 usec
Module:              dmperfss.exe
Source file:         (None)
Measurement:         Function
Has hidden functions: No
```

Descendant	Percent	Calls	Propagated Time
RegQueryValueExW	67.42	15350	31532667.57
HeapReAlloc	30.69	11511	14353861.25
HeapAlloc	1.77	3839	825453.99
HeapSize	0.11	15350	52833.77

Figure 8.

instrumented program. With VQ, you can see deeply into other processes. In this instance, at least, we found the insight we gained into the way these system interfaces work was extremely valuable. They gave us information that is simply unavailable from any other documented source.

Overall, we gave the Rational Visual Quantify program very high marks in usability. It extends the execution time analysis so that you can look inside many system calls and interfaces. The user interface is very well designed to allow the user to cope with all the additional information that is presented.

Intel VTune.

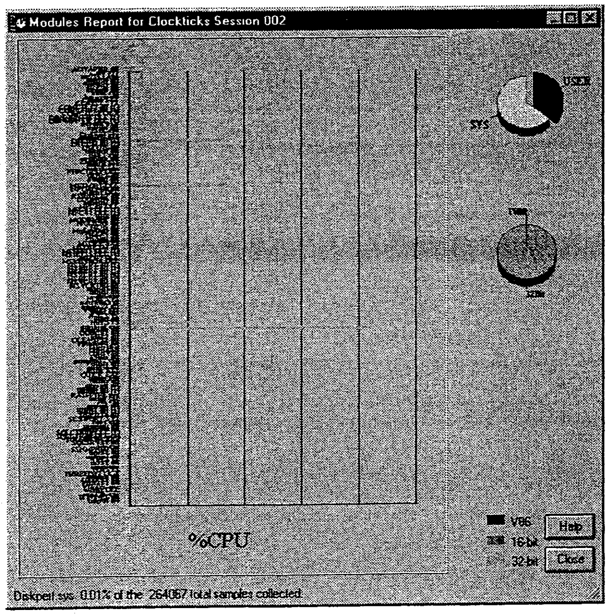
Intel's VTune product is a CPU execution profiling tool aimed at helping the programmer create code that is optimized for Intel hardware. VTune samples system-wide activity using a methodology that runs off the hardware's timer interrupts. During clock interrupt processing, VTune figures out what process was executing immediately prior to the interrupt occurring. Then VTune maps the Program Counter for the thread that was running into the process virtual address space to figure out what module and code is being executed. Later, when it has finished collecting data, VTune post processes the information to produce a very detailed report on just about every routine that is running on an NT system. The only NT system routines that are not visible to VTune are other Interrupt Service Routines and Deferred Procedure Calls.

There are several distinct advantages to this approach. First of all, this approach generates very low overhead. We made our first run with VTune using the same set up that we used under VQ – the `dmperfss` application was set to collect the default Collection set once per second. During this run, VTune identified that 90% of the system activity occurred in a module called `Hal.dll`. HAL is the hardware-specific NT Hardware Abstraction Layer. Apparently, this is where the code for the system idle thread is situated that NT dispatches when there is no other work for the system to perform. With the default Collection set being harvested once per second, the system was cruising at less than 10% busy with VTune running.

So another advantage of VTune is that it sees all system activity, even into the NT operating system kernel and the HAL. Because these operating system modules are not shareable DLLs, they are not visible to tools like Visual Quantify. Finally, this measurement methodology represents the only way to get good performance data on more complex applications, involving, for instance, multiple process address spaces, interprocess communication, etc. In other words, VTune may be the best way to gain insight into Web site applications that interact with back-end databases, or COM-based applications that run within the context of the Microsoft Transaction Server.

In the study performed here, reacting to the initial results reported by VTune (as described above), we decided it would be helpful to try and accumulate more samples related to `dmperfss` activity and less involving the NT idle thread. Consequently, we changed the collections parameters for `dmperfss` to collect all available performance Objects once per second. Remember that under VQ it was impractical to profile the program under those circumstances; but using Vtune, this scheme worked just fine.

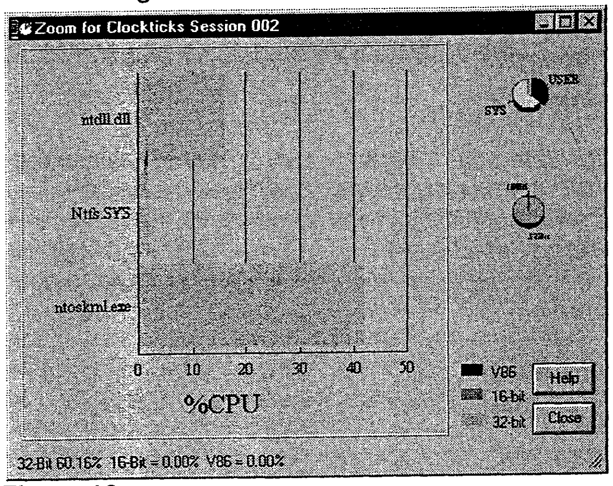
Figure 9 illustrates the VTune output reports that are available. A Modules Report is shown from an interval where `dmperfss` was executing, collecting the full Master Collection set once per second during an approximately five minute interval. All the modules detected in execution during that interval are listed on the left side of the chart in alphabetical order. The chart is a histogram showing the amount of CPU time spent inside each module. Each vertical bar represents 10% CPU consumption.



Diskperf.sys: 0.01% of the 264067 total samples collected

Figure 9.

As can be seen, this VTune report is so detailed that it is quite hard to read. By clicking on a section of the chart, you can zoom in to see more detail. See Figure 10. There are no other sort options for this display, which also contributes to the difficulty in manipulating the display. Figure 10 zooms in on the portion of the chart in Figure 9 where it just so happens that two or three modules that are consuming a good deal of CPU time sort together. These modules are NT operating

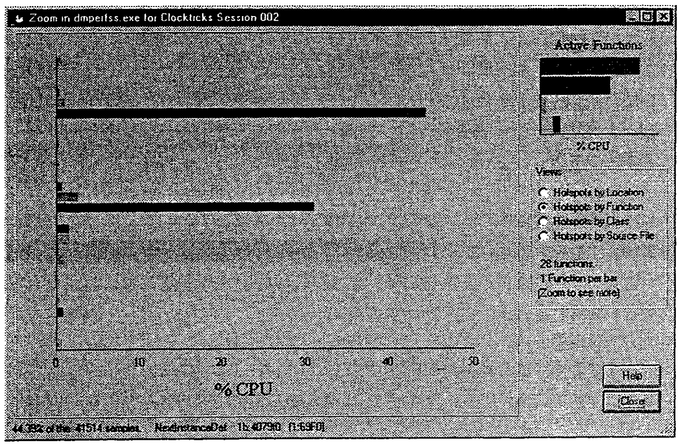


32-Bit 60.16% 16-Bit = 0.00% V86 = 0.00%

Figure 10.

system routines, `ntdll.dll` and `ntoskrnl.exe`. Neither of these operating systems modules was visible using any of the other profiling tools.

Figure 11 illustrates the ability in VTune to zoom in on specific modules. This is a picture of the `dmperfss` executable in virtual storage, showing the code addresses where VTune detected activity, which are then mapped to program modules. Here the interface



44.39% of the 41514 samples NextInstanceDef 1b:40790 [1:69F0]

Figure 11.

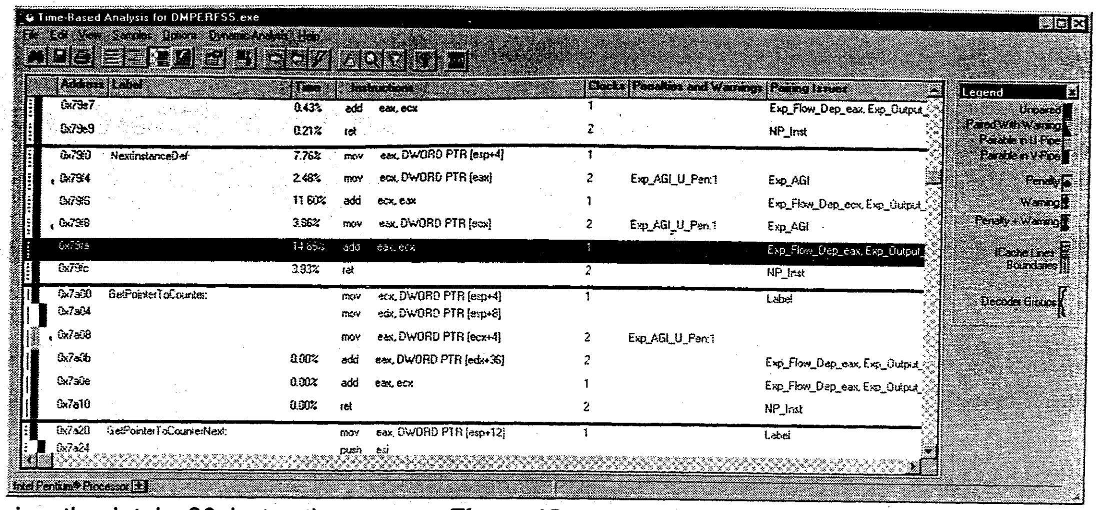
is a bit more flexible, with various sort options available for program hot spot analysis. In this view, modules are sorted by relative address in the load module.

The VTune hotspot analysis identified two functions inside `dmperfss` that accounted for more than 70% of the activity inside the process address space. The two modules VTune identified were `NextInstanceDef` and `IsPreviousAndParentSameInstance`. These Vtune results correlate well with the original MS C++ profiler and VQ runs. MS C++ found high activity inside `NextInstanceDef` and `FindPreviousObjectInstanceCounter`, the parent function that calls `IsPreviousAndParentSameInstance` internally. (Reference Figure 1.) VQ also identified `IsPreviousAndParentSameInstance` as one of the most heavily utilized modules with an extremely high number of functions calls. (Reference Figure 5).

Figure 12 shows the detailed analysis VTune performed on the code being executed in one of the modules under consideration, `NextInstanceDef`. Here execution time is mapped to the actual machine code that was generated by the C++ compiler. This is the area in which VTune stands above the crowd. There are six machine instructions associated with this routine, accounting for fully 44.39% of the CPU time consumed by the process. You can see how VTune breaks down the CPU consumption instruction by instruction. Interpreting the report output in Figure 12, we discovered, requires becoming more familiar with the Pentium hardware and its performance characteristics.

Intel Pentium hardware. VTune, we discovered, is targeted specifically for programs that need to execute on Pentium hardware. The program provides an analysis of program execution behavior on the Pentium that is very detailed and informative. It turns out that this analysis is not as useful for programs executing on newer Intel hardware, such as the

Pentium Pro or Pentium II. However, learning how to use this detailed information means understanding quite a bit about the way the Pentium (and Pentium Pro) processor chips work.



Time-Based Analysis for DMPERFSS.exe

Address	Label	Time	Instructions		Clocks	Penalties and Warnings	Pairing Issues
0x79e7		0.43%	add	eax, ecx	1		Exp_Flow_Dep_eax, Exp_Output_
0x79e9		0.21%	ret		2		NP_Inst
0x79f0	NextInstanceDef	7.76%	mov	eax, DWORD PTR [esp+4]	1		
0x79f4		2.48%	mov	ecx, DWORD PTR [eax]	2	Exp_AGI_U_Pen:1	Exp_AGI
0x79f6		11.60%	add	ecx, eax	1		Exp_Flow_Dep_ecx, Exp_Output_
0x79f8		3.86%	mov	eax, DWORD PTR [ecx]	2	Exp_AGI_U_Pen:1	Exp_AGI
0x79fa		14.85%	add	eax, ecx	1		Exp_Flow_Dep_eax, Exp_Output_
0x79fc		3.83%	ret		2		NP_Inst
0x7a00	GetPointerToCounter:		mov	ecx, DWORD PTR [esp+4]	1		Label
0x7a04			mov	edx, DWORD PTR [esp+8]			
0x7a08			mov	eax, DWORD PTR [ecx+4]	2	Exp_AGI_U_Pen:1	
0x7a0b		0.00%	add	eax, DWORD PTR [edx+36]	2		Exp_Flow_Dep_eax, Exp_Output_
0x7a0e		0.00%	add	eax, ecx	1		Exp_Flow_Dep_eax, Exp_Output_
0x7a10		0.00%	ret		2		NP_Inst
0x7a20	GetPointerToCounterNext:		mov	eax, DWORD PTR [esp+12]	1		Label
0x7a24			push	esi			

Figure 12.

The Pentium is a fifth generation microprocessor running the Intel x86 instruction set. Hardware designers [see, for example 6] refer to the Intel x86 as a CISC (Complex Instruction Set Computer), a style of hardware that is no longer in style. Today, hardware designers generally prefer processor architectures based on RISC, Reduced Instruction Set Computers. The complex Intel x86 instruction set is a legacy of design decisions made twenty years ago at the dawn of the microprocessor age when RISC concepts were not widely recognized. The overriding design consideration in the evolution of the Intel x86 microprocessor family was maintaining upward compatibility of code that was developed for the original 8086 machines.

Processor	Year	Clock Speed (MHz)	Bus Width (bits)	Addressable Memory	Transistors
8080	1974	2	8	64K	6,000
8086	1978	5-10	16	1 MB	29,000
8088	1979	5-8	8	1 MB	29,000
80286	1982	8-12	16	16 MB	134,000
386 DX	1985	16-33	32	4 GB	275,000
486 DX	1989	25-50	32	4 GB	1,200,000
Pentium	1993	60-233	32	4 GB	3,100,000
Pentium Pro	1995	150-200	64	4 GB	5,500,000
Pentium II	1997	233-333	64	4 GB	7,500,000

Figure 13.

Figure 13 summarizes the evolution of the Intel x86 microprocessor family starting with the 8086, first introduced in 1978. As semiconductor fabrication technology advanced and more and more transistors were available to the designers, Intel's chip designers added more and more powerful features to the microprocessor. For example, the 80286 (usually referred to as the 286) was a 16 bit machine with a form of extended addressing using segment registers. The next generation 386 chip maintained compatibility with the 286's rather peculiar virtual memory addressing scheme, while it implemented a much more straightforward 32-bit virtual memory scheme. In contrast to the 16-bit 64K segmented architecture that was used in the 286, the 386 virtual addressing mode is known as a "flat" memory model.

The extra circuitry available in the next generation 486 processors introduced in 1989 was again used to create a higher performance chip with more built-in features. The 486 microprocessor incorporated floating point instructions (available in an optional coprocessor during the days of the 386) and a small 8K level 1 code and data cache. Including a small cache memory meant that the 486 could also speed up instruction using pipelining. Pipelining is a familiar processor speed up technique that attempts to exploit an inherent parallelism in the process of decoding and executing computer instructions. The 486 breaks instruction execution into five stages, as illustrated in Figure 14:

- Prefetch,
- Instruction Decode, part 1 (op code interpretation),
- Instruction Decode, part 2 (operand fetch)
- Execute, and
- Write back, (Registers and memory are updated).

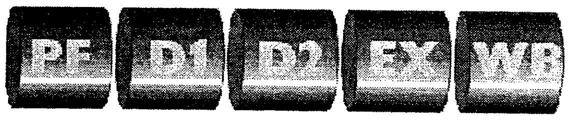


Figure 14.

Since an instruction will spend at least one clock cycle in each stage of execution, a 486 instruction requires a minimum of five clock cycles to execute. Pipelining is where the Level one processor cache memory comes into play. The microprocessor can access a line of data in cache in one clock cycle, while there is a significant performance penalty if data or instructions actually have to be fetched from main memory. The fact that the Intel uses CISC is also a significant performance factor because many 486 instructions require more than one clock cycle in the Execute stage. (The significance of pure RISC designs is that only simple instructions that can execute in a single clock are implemented.)

There are separate pieces of hardware circuitry in the processor that are responsible for carrying out the processing associated with each stage in the execution of a machine instruction. For instance, after an instruction reaches the D1 stage, the circuitry associated with fetching the next instruction is idle. The idea behind a pipeline is to utilize this hardware by attempting to execute instructions in parallel. As illustrated in Figure 15, the 486 pipeline has the capacity to execute five instructions in parallel. As soon as Instruction 1, for example, completes its Prefetch stage, the prefetch hardware can be applied to the next instruction in sequence. When the 486 pipeline is working optimally, even though each individual instruction takes five clock cycles to executes, on average an instruction completes *every clock cycle*! The behavior of a pipelined processor architecture leads quite naturally to measuring its performance according to the number of Clocks per Instruction (CPI). Pipelining boosts the actual instruction rate of a microprocessor from 5 CPI for the non-pipelined version to 1 CPI for the ideal processor.

In practice CPIs in the range of one clock per instruction are not achievable even in the very best textbook examples of RISC processor design. Some of the problems are endemic to the technology. Various instruction sequences result in pipeline stalls that slow down instruction execution rates. Branch instruction which change the sequence of instruction execution are problematic because the wrong instructions get loaded and decoded when the branch that changes the sequence is executed. Processors like the Pentium and Pentium Pro use *branch prediction* strategies to keep track of when branches were taken in the past and load the pipeline with instructions out of sequence based on history. Often it is necessary to stall the pipeline because the output from one instruction is required by the next instruction. When one instruction updates a register and the instruction that follows uses that register to address data, it is necessary to stall the pipeline in the address generation stage for the second instruction. This type of a pipeline stall where there is a dependent relationship between instructions that execute near each other is known as an *interlock*.

Intel's experience with speeding up the 486's instruction execution rate using pipelining was disappointing for a different reason – the x86 complex instruction set. Complex x86 instructions require more than one clock cycle in the execution stage. Reviewing the specifications in Intel's documentation[7], we can see that the basic commands in the integer instruction set require between 1 and 9 clock cycles. The `rep` prefix used in the commonly used bulk memory Move instructions alone requires 4 clocks, for example. A 32-bit far call, used for branching to and from a subroutine, can require as many as 22 clock cycles. This variability in instruction execution time plays havoc with the 486's 5 stage pipeline, causing frequent stalls in the EX stage, as depicted in Figure 16. The drawing illustrates a 486 pipeline stall because Instruction 1's EX cycle requires five clocks to complete. You can see how a stall in one instruction backs up the entire pipeline. Because some its complex instructions require many clock cycles to execute, the 486 sustained instruction execution rates that fell well short of optimal performance.

As the next generation semiconductor fabrication technology became available, Intel's chip designers faced a quandary. Some pipelining performance issues can be addressed with more hardware, so the P5 or Pentium chip gained separate code and data caches, as well as branch prediction logic. (The Pentium's use of branch prediction was subject to a well-publicized patent infringement suit brought by Digital. The litigation was settled out of court in 1998.) But the performance issues related to the x86 complex instruction set resisted a simple hardware solution.

The Pentium introduces a superscalar dual pipeline architecture that allows, under the right circumstances,

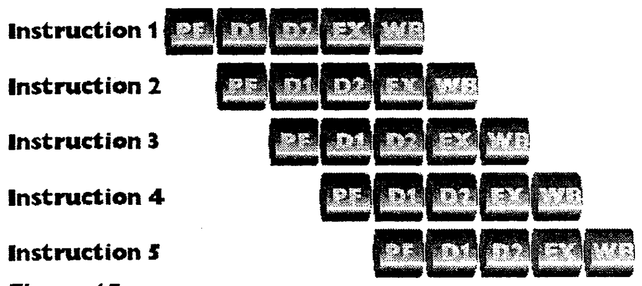


Figure 15.

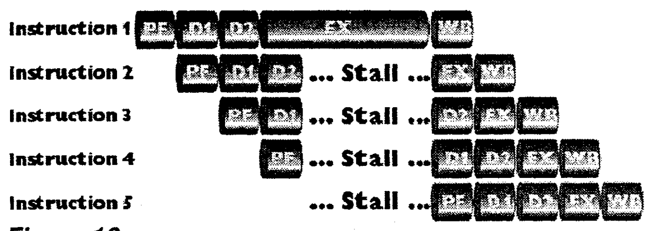

Instruction 1
PF D1 D2 EX WB
Instruction 2
PF D1 D2 ... Stall ... EX WB
Instruction 3
PF D1 ... Stall ... D2 EX WB
Instruction 4
PF ... Stall ... D1 D2 EX WB
Instruction 5
... Stall ... PF D1 D2 EX WB

Figure 16.

two instructions to be completed in a single clock cycle. The Pentium dual pipeline is illustrated in Figure 17. The Pentium contains a single PreFetch engine capable of operating on several instructions in parallel.

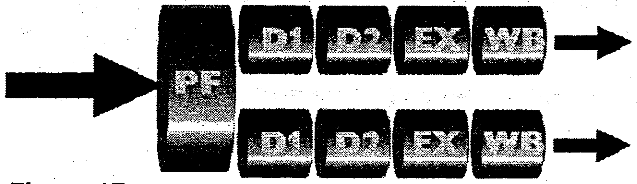


Figure 17.

The Pentium can then load the two instruction execution pipelines in parallel. Where a pipeline computer is theoretically capable of executing instructions at a rate of 1 CPI, a superscalar machine such as the Pentium is capable of an instruction execution rate that *is less than* 1 CPI. The top pipeline in the Pentium superscalar pipeline is known as the *u pipe*, and its execution characteristics are identical to the 486. The bottom pipeline is called the v pipe and it is only loaded under special circumstances. The Pentium Prefetch stage follows arcane *instruction pairing rules* that determine whether the second instruction in an instruction pair is loaded in the v pipe and executed in parallel.

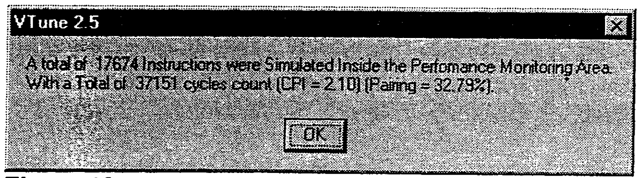
The rules for loading the v pipe are fairly complex. Two simple single cycle execution instructions can almost always be paired and executed in parallel. To some observers [see 8, for example], this characterizes the Pentium instruction pairing rules as identifying a subset of RISC instruction inside the full, complex instruction set and allowing them to execute in parallel. However, any instructions which use immediate operands or addresses (the data the instruction operates on is embedded in the instruction) can never be paired. The pairing rules are more complicated than that, however. For example, if the second instruction operates on any of the same registers as the first instruction, it cannot be executed in parallel. This is a particularly strict requirement in the x86 environment where there is a legacy of only eight General Purpose Registers, which leads to a few registers being used over and over.

Superscalar architectures were introduced into the world of scientific computing in the late 1980s by workstation hardware manufacturers who also developed their own systems software, including the compilers that generated code optimized to run on these machines.[7] RISC hardware manufacturers rely on compilers to generate code that will run optimally in a parallel environment to take full advantage of the superscalar architecture. An optimizing compiler may resort to inserting placeholder instructions into the instruction sequence or rearranging instructions to avoid sequences where there are direct dependencies between successive instructions.

In the open PC environment, Intel's holds an enviable position as the developer of the hardware used in most PC desktop, workstation, and Server machines. However, Intel develops very little of the systems software that run on its hardware, including the most popular operating systems and high level language compilers. Intel's challenge when it introduced the Pentium supercalar architecture was to promote the use of this hardware among third party systems software developers, including the developers of compilers and operating systems.

Intel's approach to promoting the Pentium was to provide two types of tools for use by third party developers. The first was to build into the processor a measurement interface that third party software could tap. The measurement interface for the Pentium provides extensive instrumentation on internal processor performance. (A complete list of available Pentium Counters is documented in Appendix A. Pentium performance Counters can be accessed under Windows NT by installing the P5 Counters, using software that Microsoft distributes as part of the Windows NT 4.0 Workstation Resource Kit[9].) It includes the ability to measure the actual instruction execution rate, the number of paired instructions that executed in the v pipe, and various metrics that deal with pipeline stalls. The hardware measurement interface allows two of the metrics listed in Appendix A to be collected at any one time.

The second tool is VTune, which performs two key functions that developers can use to optimize the code they develop. The first is it provides a very usable interface to the built-in Pentium measurement interface. Using this interface, VTune can be used to collect Pentium statistics on a program as it executes. The second key aspect of VTune is the capability to analyze code sequences and make recommendations on how to write code that is optimal for the Pentium. Among other things, VTune computes the CPI for an instruction sequence and calculates the utilization of the v pipe. See Figure 18 below.



A total of 17674 Instructions were Simulated Inside the Perfomance Monitoring Area.
With a Total of 37151 cycles count [CPI = 2.10] [Pairing = 32.79%].

Figure 18.

With this background, we can know return to VTune's analysis of the `NextInstanceDef` subroutine that was identified as a hotspot within the dmperfss program, as depicted back in Figure 12.

The Microsoft Visual C++ compiler generated the six lines of machine code for the subroutine from the following C language statements:

```
PERF_INSTANCE_DEFINITION * NextInstanceDef
            ( PERF_INSTANCE_DEFINITION *pINSTANCE )
{
    PERF_COUNTER_BLOCK  *pCtrBlk;
    pCtrBlk = (PERF_COUNTER_BLOCK *)
       ((PBYTE)pInstance + pInstance->ByteLength);
    return     (PERF_INSTANCE_DEFINITION *)
               ((PBYTE)pInstance + pInstance->ByteLength + pCtrBlk->ByteLength);
{
```

What this C language helper function does is advance a pointer inside the buffer of data returned by RegQueryEx from the beginning of one Object instance to the next occurrence of an instance. When dmperfss is retrieving instanced data, particularly associated with NT processes and threads, this code is called repeatedly to parse the performance data buffer. As we have seen, all three performance profiler products identified this segment of code as an execution hotspot in the program. In the VTune analysis of dmperfss, the NextInstanceDef code segment was executed even more frequently because both process and thread data was being collected. The profiling data strongly suggests that the efficiency of the program can be optimized by improving the performance of this specific segment of code.

The code generated by the Microsoft compiler which carries out these C langauge statements is a sequence of admirably compact machine language instructions:

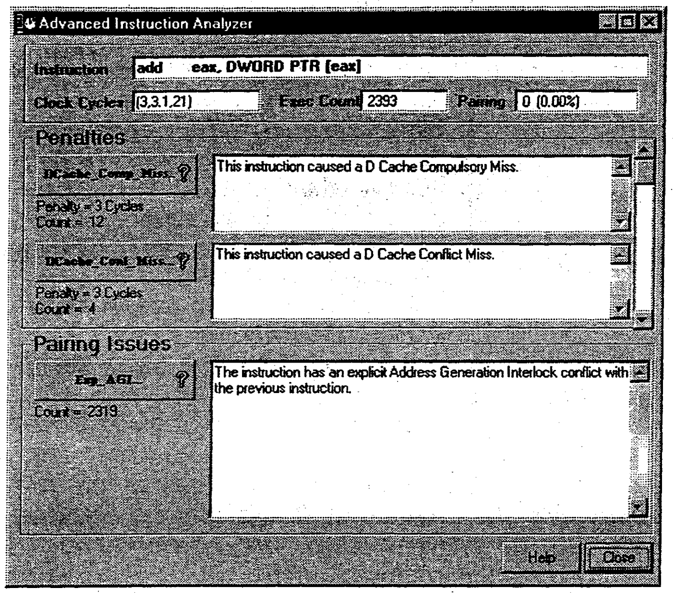
```
00408D40      mov       Eax,dword ptr [esp+4]
00408D44      mov       ecx,dword ptr [eax]
00408D49      add       ecx,eax
00408D46      mov       eax,dword ptr [ecx]
00408D4B      add       eax,ecx
00408D4D      ret
```

The code analysis VTune performs on the machine language instructions in NextInstanceDef (illustrated in Figure 12) indicates that *none* of these frequently executed instructions is capable of being executed in parallel on the Pentium. The total lack of parallelism comes despite the fact that these are all simple one and two cycle instructions. The screen legend in the right hand corner of the VTune display is used to decode the visual clues the program provides to instruction execution performance. Instructions which can be successfully paired and executed in parallel are clearly indicated, as are the boundaries of code cache lines. The P5 optimization switch on the VC++ compiler generates NO OP instructions to line up code on cache line boundaries, as shown here.

Not only is this code unable to take advantage of the Pentium's parallelism, VTune informs us that the machine code instructions generated by the compiler stall the u pipe. The column marked "Penalties and Warnings" indicates that the second and fourth MOV (move) instructions cause an address generation interlock (AGI) that stalls the u pipe. Notice that each instruction in this routine is executed once and only once each time through the routine. There are no branches. However, the instruction timing VTune reports, based on its sampling of the program as it was running, show a wide variation in the execution time of the individual instructions in this code sequence.

The instruction execution timings VTune reports clearly show the performance impact of stalling the pipeline. The second MOV instruction, requiring two clock cycles to execute, is found in execution 2.48% of the time. This instruction copies the value at the address pointed to by the EAX register into the ECX work register. The previous instruction sets up the EAX address using a parameter passed on the stack pointer (ESP). There is an obvious dependency between these two instructions. The next instruction adds a value to ECX. The code is doing arithmetic on another address pointer and uses this value in the MOV instruction that follows. Because the first MOV stalls the pipeline, the ADD instruction that follows is found to be in execution 11.6% of the time. Continuing the analysis, we see how pipeline stalls propagate through the an instruction sequence. The next MOV instruction (another 2 cycle instruction) is in execution 3.86% of the time, while the single one clock ADD that follows it was found to be in execution 14.86% of the time!

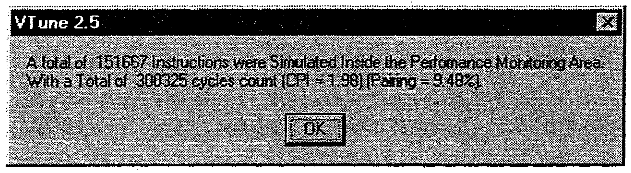
Faced with this situation, a programmer working in Assembly language can rework the machine instructions easily enough to avoid the address generation interlock problem by adding a third work register. The more complicated code sequence actually runs several times faster than the original. VTune provides detailed advice to the Assembly language programmer concerning Pentium-specific instruction execution performance issues, as illustrated in Figure 19. The ADD instruction analyzed below has an obvious problem due to the interlock with the previous instruction. But it also occasionally requires the data referenced by the EAX register to be refreshed from memory, rather than using the copy that was found in the data cache.



```
Advanced Instruction Analyzer
Instruction   add    eax, DWORD PTR [eax]
Clock Cycles  (3,3,1,21)    Exec Count 2393    Pairing 0 (0.00%)
Penalties
DCache_Comp_Miss   Penalty = 3 Cycles  Count = 12
  This instruction caused a D Cache Compulsory Miss.
DCache_Conf_Miss   Penalty = 3 Cycles  Count = 4
  This instruction caused a D Cache Conflict Miss.
Pairing Issues
Exp_AGI   Count = 2319
  The instruction has an explicit Address Generation Interlock conflict with
  the previous instruction.
```

Figure 19.

To take advantage of all the detailed performance information on instruction execution that VTune provides, an application programmer working in a C language development faces three choices. The first and easiest option is to inform the compiler to generate code optimized for the Pentium processor. Using the P5 optimizing switch, we noted some changes in the sequence of instructions generated for this routine, but nothing that was an extensive enough restructuring of the program logic to show any appreciable improvement. Figure 20 summarizes a run we made after instructing the compiler to generate code optimized for the Pentium. The CPI shows a slight reduction compared to Figure 17, although, curiously, the percentage of paired instruction execution actually dropped. Of course, CPI is the more important indicator of performance.



```
VTune 2.5
A total of 151657 Instructions were Simulated Inside the Perfomance Monitoring Area.
With a Total of 300325 cycles count [CPI = 1.98] [Pairing = 9.48%]
```

Figure 20.

The second option is to replace the code generated by the compiler with an inline Assembly language routine. We did experiment with that option in this instance and were very satisfied with the performance improvements that resulted. This was in spite of the fact that adding another work register make the code longer and somewhat more complicated. This counterintuitive outcome is not unusual for RISC machine. They can often execute longer code sequences faster than shorter, more compact ones. The third option is to recode the original C language routine, which is the route we believe is indicated in this instance to address the number of times this helper function is being called. We plan to tackle that code restructuring in the next development cycle, and we will continue to rely on Rational Visual Quantify and Intel VTune to measure the impact of those improvements.

Intel Pentium Pro. A final topic that should be addressed before we leave a discussion of the capabilities of VTune behind is the role of VTune with Intel's current generation of Pentium Pro and Pentium II hardware. The P6 or Pentium Pro internal structure is much more complicated than the previous generation P5. The simple parallel u and v pipes are replaced by a complex *microarchitecture* that addresses x86 instruction execution issues from an entirely different direction. As the discussion above indicated, instruction pairing in the Pentium was limited to a relatively small subset of the available instruction set. To get the most mileage out of the Pentium hardware, compilers had to be revised, and C language programs that ran on the 486 had to be recompiled for the P5. Of course, being forced to maintain two separate and distinct sets of load modules, one optimized for the 486 and the other optimized to run on the Pentium, is problematic, to say the least. But without recompiling older programs to make more use of the simple RISC-like instructions that can be executed in parallel on the Pentium, Intel customers will not reap the full benefits of the new and potentially much faster hardware. A second issue in generating optimal code for the Pentium was the problem illustrated here. Having very few General Purpose registers to work with means it is difficult to write x86 code that does not stall the pipeline and can take full advantage of the superscalar functions.

The microarchitecture devised for the P6 attempts to address these specific performance issues. The P6 has even more parallel processing capabilities than the P5. (See 10 for a detailed discussion of P6 internals.) First, the P6 automatically translates x86 instructions into a RISC-like set of fixed length micro-operations or *micro-ops*. Most micro-ops are designed to execute in a single clock cycle. To augment the limited number of GPRs available to the machine language programmer, these micro-ops can draw on 40 internal work registers. After unwinding CISC machine instructions into RISC-like instructions, micro-ops are stored in a pool where they can be executed in any order by the dispatch unit of the processor. When *all* the micro-ops associated with a given machine language instruction complete execution, the instruction itself is said to be *retired.* Retiring instructions also means that as any results are written back to the computer's registers and memory. The processor's retirement unit can retire up to three micro-ops per clock cycle, with the restriction that these must be in strict order according to the original instruction execution stream.

By design, the P6 microarchitecture is designed to create automatically the sort of optimized code sequences that the programmer needed to craft by hand using VTune. As a matter of fact, VTune cannot perform the same kind of analysis illustrated here on code instruction sequences for the much more complicated P6. How effective the complex P6 architecture actually is in optimizing code sequences remains an open question. The profile data collected by VTune on the performance of the `dmperfss` application that was reported in Figure 12 was collected on a Pentium Pro running at 200 MHz. From these results, it is evident that the P6 did not parallelize this instruction sequences very successfully. The P6 microarchitecture apparently could not eliminate the pipeline stalls caused by address generation interlock in this code sequence. This is a clear indication that VTune continues to play a valuable role in optimizing the performance of Wintel applications.

References.

[1] Chris Loosely and Frank Douglas, *High-Performance Client Server*. New York: John Wiley and Sons, 1998.
[2] Connie U. Smith, *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley, 1990.
[3] Jeff Buzen and Annie Shum, "Considerations for Modeling Windows NT," CMG97 *Proceedings*, 1997, p. 219-229.
[4] Steven E. Sipe, "C++ Code Profilers," PC Magazine, October 21, 1997.
[5] Ron van der Wal, "Source-code profilers for Win32," *Dr. Dobbs Journal*, March 1998.
[6] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kauffman, 1996, 2nd Edition.
[7] *Addendum - Intel Architecture Software Developer's Manual: Volume 2: Instruction Set Reference*. http://developer.intel.com/design/mmx/manuals/.
[8] Hans-Peter Messmer, *The Indispensable Pentium Book*. Reading, MA: Addison-Wesley, 1995.
[9] *Microsoft Windows NT Workstation Resource Kit*. Redmond, Washington: Microsoft Press, 1996.
[10] Tom Shanley, *Pentium Pro Processor System Architecture*. Reading, MA: Addison-Wesley, 1997.

Appendix A. Pentium (P5) performance Counters

% Branch Target Buffer Hits
% Branches
% Code Cache Misses
% Code TLB Misses
% Data Cache Misses
% Data Cache Read Misses
% Data Cache Write Misses
% Data Snoop Hits
% Data TLB Misses
% V-Pipe Instructions
Bank Conflicts/sec
Branches/sec
BTB Hits/sec
Bus Utilization (clks)/sec
Code Cache Miss/sec
Code Read/sec
Code TLB Miss/sec
Data Cache Line WB/sec
Data Cache Snoop Hits/sec
Data Cache Snoops/sec
Data R/W Miss/sec
Data Read Miss/sec
Data Read/sec
Data Reads & Writes/sec
Data TLB Miss/sec
Data Write Miss/sec
Data Write/sec
Debug Register 0
Debug Register 1
Debug Register 2
Debug Register 3
FLOPs/sec
I/O R/W Cycle/sec
Instructions Executed In vPipe/sec
Instructions Executed/sec
Interrupts/sec
Locked Bus Cycle/sec
Memory Accesses In Pipes/sec
Misaligned Data Refs/sec
Non_Cached Memory Ref/sec
Pipe Stalled On Addr Gen (clks)/sec
Pipe Stalled On Read (clks)/sec
Pipe Stalled On Writes (clks)/sec
Pipeline Flushes/sec
Segment Loads/sec
Stalled While EWBE#/sec
Taken Branches or BTB Hits/sec
Write Hit To M/E Line/sec