Source Code Instrumentation and its Perturbation Analysis in Pentium II

Haleh Najafzadeh and Seth Chaiken

May 22, 2001

1 Abstract

Microprocessors typically have software readable counters for events such as instructions executed, cycles, instruction stalls, and cache misses. Besides their usefulness to count overall performance metrics, these counters can be used to reveal details about dynamic process behavior and hardware effects of compiler optimizations, but interference between probe code and code under test makes it difficult to interpret the counts accurately for pure application code. In this paper the problems to reduce instrumentation perturbations, to model and correct for them and then quantify the remaining uncertainties are addressed for the Pentium II. After having eliminated most operating system and interrupt interference by embedding benchmarks into Linux kernel modules, we introduce the "null instrumentation" strategy in which code under test is displaced but not removed. We report as remaining uncertainty levels the differences between results of "full" (detailed) instrumentation and

results of "raw" (global) instrumentation. This technique is validated against static instruction count analysis. Relative remaining uncertainty levels in most event counts for a typical image processing code are less than 4%, with many below 1%.

2 Introduction

A program profile attributes run-time costs to portions of a program's execution. Profiling systems fall into two groups:

Group 1 (Counting profiler): Source code instrumentation, binary modification, and/or compiler support are used to gather measurements. *gprof[11]*, *gcov*[23], *pixie*[31], *jprof*[25], *quartz*[3], *MTOOL*[10], *SimOS*[26], Intel's VTune *dynamic analyzer*[12], part of SGI's *SpeedShop*[34], part of *Sprof*[17], *Rabbit*[16], *EEL*[19].

Group 2 (Statistical profiler): The program or some other counter is periodically sampled, typically after timer interrupts. Examples include *Morph*[35], *iprobe*[2], *prof*[23], part of *SGI's SpeedShop*, part of *Sprof*, *DCPI[2]*.

Some Group 1 tools (gprof, gcov, pixie) apportion and summarize simple metrics, such as execution frequency or elapsed time, to static syntactic units, such as procedures, statements or instructions. Other tools in this group (Intel's Vtune, part of Speedshop, part of Sprof, Rabbit, EEL) report more hardware related information but their papers or documentation do not give detailed overhead analysis. Those Group 2 tools that don't enable source-level instrumentation do not allow their users to supply semantic information nor to concentrate on detailed behavior of frequently executed code. For instance such limitations of SGI's Speedshop are reported by [29]. Also [2] reports that iprobe, the VTune sampler, and

SpeedShop are unable to map the sample data accurately back to individual instructions.

Our overall objective is to study computer behavior and its hardware features by observing the low level interactions of a program with hardware, especially the effects of some machine instruction executions upon the performance factors of other instruction executions. One application is to predict the performance of an original program from an instrumented execution with a model to compensate for overhead of direct and indirect perturbations (see section 4). We extend models like those of [22] to the Pentium and to metrics beyond execution time (see section 5). All but a few ([22, 6], e.g.) prior studies and tools do not account for overhead. A second application is to test assumptions about our hardware's organization and dynamic behavior.

We seek full control and selectivity of hardware counter measurements with quantified overheads. We then seek to minimize the overhead, develop a model to estimate performance from the instrumented execution output as in [22], and to validate the estimates and error analysis. Existing tools would not give us the overhead analysis and would require us to use more than one tool to obtain the hardware related information we wanted. For this reason we started from the scratch to explore the intrinsic capabilities of the Pentium II programmable hardware counters. We chose as did [29] source code level instrumentation for simplicity, efficiency and future portability concerns (see section 3).

We began to develop methodologies to handle optimized real application code and to quantify the limitations that compiler optimization imposes on monitoring. On the other hand, we eventually moved our benchmarks into the kernel to eliminate the overwhelming interference from interrupts and other operating system activity [22, 2, 6]. Hence, to apply our methodologies to real applications on a commodity operating system and hardware platform, we are seeking to develop more support for kernel-application data communication.

Finally, we validated the accuracy of various performance counter reports by careful experiment design, first validated with instruction count on small codes. Instruction count is independently predictable from static analysis. [18] recommends validating one model with at least one other independent model or experiment. Accuracy of other metrics after compensation is evaluated by comparing a "raw" (non-instrumented) measurement with its prediction from instrumented execution. Our numerical results appear in Table 1 at the end of section 5.

In this paper, "instrumentation" denotes the program after modification by inserting "probe" operations to capture hardware counter values. The Pentium Pro and MMX introduced the user mode RDPMC instruction which enables user mode programs to read the counters without operating system call overhead. We wrote the probe code in GNU C extension `asm()` statements[30, 13] to insert RDPMC and other assembly language code into C programs. We also developed convenient device drivers to select and activate the counters since the necessary Pentium MSR instructions are in kernel mode (ring 0).

3 Methodology

In this section, we document the hardware and software details of our experiments.

3.1 Experimental Environment

Since our goals include real applications, we extracted our simplest two benchmarks from the blurring plug-in of the Gimp image editing and processing software (unstable developer's version 1.1.13 [15]). This version allowed us to run the plug-in from the command line. Our operating environment is the Mandrake 6.5 distribution of Linux with kernel version 2.2.13-4mdk and gcc compiler version 2.95.1 The hardware is a 400 MHZ Pentium II, with L2 cache size of 256 KB, and 64MB of memory. The processor includes separate data and instruction L1 caches. The on-chip level one (L1) caches consist of one 8-Kbyte four way set associative instruction cache unit and one 8-Kbyte two-way set associative data cache unit. Both instruction and data L1 cache line sizes are 32 bytes.

3.2 Hardware Performance Counters

The Intel Pentium, Pentium Pro and later processor have one 64 bit time stamp (CPU cycle) counter register and two 40-bit software-configurable performance counters which allow two types of events to be counted simultaneously. The RDTSC instruction copies the current value of the time-stamp counter into the EDX:EAX registers. The RDPMC instruction copies the current value of the counter specified by ECX into the EDX:EAX registers. RDTSC and RDPMC instructions are legal in user mode[1]. Model specific performance event select registers (PerfEvtSel0 and PerfEvtSel1) and performance counters (PerfCtr0 and PerfCtr1) are accessed with the privileged RDMSR and WRMSR instructions to program and reset the counters. Therefore, it is necessary to write some code that runs in kernel mode. For maximum flexibility and ease of installation, we wrote a loadable device driver module rather than modifying the booted kernel. See section 10.6.1 of [8] for PerfEvtSelx programming details and appendix A of [7] for the description of the events available for counting.

[1] after a kernel mode instruction sets a model specific register bit.

3.3 Optimized Instrumentation

In this section we will describe a few steps we have taken in order to minimize the instrumentation perturbation. Some steps resemble the instruction schedule design in the executable file edit technique for profiling of [28].

Data structure design to improve locality of reference

We replaced two integer arrays with one array of integer pairs for capturing the EDX:EAX values from RDPMC and observed a reduction in the number of cache misses as predicted by [24] page 405. For example, preliminary Null instrumentation tests with only one pair of non-optimized probes showed the L1 Dcache miss count was reduced from 3 to 1, and the data reference count was reduced from 12 to 9.

Choice of the static C storage class to guarantee alignment and reduce addressing overhead

Declaring the array of integer pairs static (rather than automatic) resolves its address at link time. We observed two improvements: 1) Fewer registers were needed for addressing. This helped reduce the code size and CPI. 2) 32-byte alignment of each array element was guaranteed, so no pair crossed a 32-byte cache line boundary. Preliminary analysis of cache stall counts showed 6-12 stall cycles were saved by better alignment.

Using more efficient Pentium formats

The GNU C language extension `asm()` statement permits C expression values or references to be substituted into assembly language operands. If we can replace a variable expression with a constant, the compiler can choose more efficient instructions.

```
__asm__ volatile ("rdpmc" : "=d" (Val_EDX), "=a" (Val_EAX):"c" (Val_ECX) );
 804bfee: 8b 0d 94 fb 04 08     movl   0x804fb94,%ecx
 804bff4: 0f 33                 rdpmc
```

has one additional memory reference and four more bytes, compared to:

```
__asm__ volatile ("rdpmc" : "=d" (Val_EDX), "=a" (Val_EAX):"c" (0) );
 804c08f: 31 c9                 xorl   %ecx,%ecx
 804c091: 0f 33                 rdpmc
```

We used keyword `volatile` to stop gcc's optimizer from moving the `asm` statement out of a loop.

Hand optimizing the probes

We optimized the probes by designing all the probe related instructions by hand rather than depending on the compiler to generate them. This way we reduced the counts for each probe from 17 to 7 instructions as can be seen below. We seek to observe hardware behavior on optimized application code. Despite full optimization, the compiler seemed to do a poor job with indexing our probe capture array. Table 1 reports the differences of the results.

```
#define pm_len 100
struct pm_record{unsigned int Val_EAX;  unsigned int Val_EDX; };
#define pm_hi_low_offset 4
struct pm_record pm_A[pm_len];
volatile struct  pm_record  *pm_P = pm_A;
/*File scope insures static extent*/
 ...
                    /*typical probe*/
                     __asm__ volatile( "xorl %%ecx, %%ecx\n\t"
                                 "rdpmc\n\t"
                                 "movl %0, %%ecx\n\t"
                                 "movl %%eax, (%%ecx)\n\t"
                                 "movl %%edx,%c1(%%ecx)\n\t"
                                 "addl $%c2, %%ecx\n\t"
                                 "movl %%ecx, %0"
                                 :
                                 : "m" (pm_P) ,
                                 "i" (pm_hi_low_offset),
                                 "i" (sizeof( struct pm_record))
                                 :"%eax","%edx","%ecx");
```

Continuous read

Every start or stop of a performance counter contributes a large overhead due to privileged RDMSR/WRMSR instructions and their preparation plus an operating system call if from user mode. Instead of starting and stopping counters repeatedly, we only start the counters once. The probes read and store successive values into an array. The count values are analyzed after data collection is completed.

4 Perturbation

Software instrumentation perturbs the state of the measured system[22]. Direct perturbations are perturbations of any performance metric that can be attributed directly to the individual probe code instructions. These instructions are the Pentium RDPMCs and extra instructions needed to use an array to acquire the counts. All other perturbations are called indirect.

Even the insertion of hand optimized probes affects the compiler's register allocation and other optimization choices made for the code under test. We classify such "compiler effects" as a sub-category of indirect perturbations. A second such sub-category is the effect of probe instruction execution on the pipeline and caches. The cache and memory traffic generated by the instrumentation could flush application data from the cache causing more overhead. A third indirect sub-category is extra context switches and external interrupts, since instrumented applications execute longer and have a larger working set. The effect of these "external perturbations" effect is known to be significant[22, 2, 6].

In order to observe hardware effects independent of "external perturbations" and effects of interrupts and operating system activity on the pipeline and caches, we ran our experiments in kernel mode. Running the entire `blur` plug-in of gimp in the kernel is difficult since the kernel has no access to the user level header files and libraries. We therefore rewrote the core computation so that it would not use gimp or C library functions. We traced the original user mode `blur` with its default values to acquire internal variable initializations. We then supplied these initial values to our blur kernel module.

To eliminate all maskable interrupts we embedded the code under test in the following sequence of Linux kernel function calls:

`save_flags(flags);` `cli();` Code under test `restore_flags(flags);`

5 Performance Perturbation Models

Formal models of performance perturbation that would permit quantitative evaluation of instrumentation costs have been explored in some extent by [22].

Our model is based on a control flow graph where nodes represent either blocks of code under test or probes. In all of our figures, we use $\bigcirc$ to represent code under test and $\square$ to represent a probe. This model is designed to apply to a variety of underlying procedural and assembly/machine languages.

5.1 Definitions

The following summarizes and discusses some definitions. See figures 1, 2, 3, 4 and 5.

Block: Arbitrary sequence of underlying language statements that could contain control operations.

Program: A program is a directed graph, where each node represents a block. Each possible control flow entry or exit of the block is represented by an in-arrow or out-arrow respectively incident to the node representing the block.

Code under test: The set of nodes which we choose to monitor.

Probe: Operation that has been inserted into or near the code under test to acquire a measurement.

Cost: The model attributes a constant or variable cost added to a performance count for each execution of each node (block). Total cost is the cost sum from all executions.

A program model can be refined if necessary by "articulating" one block into finer blocks. We had found articulation necessary to model compiler perturbation effects under our original probe implementation. When hand optimized probes were used, the instruction level interference between compiler generated probe support code and certain code under test disappeared.

Instrumentation: The program after modification by insertion of probes[2].

Full instrumentation: Covering each block of the code under test with a pair of probes, one before and one after the block.

Raw instrumentation: Covering the entire code under test with a single pair of probes.

Null instrumentation: Null instrumentation is constructed by changing the position of the probes so probe pairs no longer cover their code under test. Note that the code under test is not deleted; it is merely displaced.

Full (resp. Raw) Null instrumentation: The Null instrumentation modification applied to the original Full (resp. Raw) instrumentation.

$\mathbf{rd}i[j]$: Within a program, $\mathbf{rd}i[j]$ denotes the operation to capture the j_th performance

[2]Supporting operations such as counter setups are designed to not interfere with the code under test and are not classified as probes.

count of the *i_th* category. This value is denoted by **rd**$i[j]$ within our analysis formulas.

RawRead: The difference between the two probe values in Raw instrumentation.

FullRead: The difference between the first and last probe values in Full instrumentation. It is the gross performance count for the code under test plus probes.

RawResult: Our estimate for the performance count of the code under test calculated from Raw instrumentation measurements.

FullResult: Our estimate for the performance count of the code under test calculated from Full instrumentation measurements.

5.2 Analytical Principles

The general methodologies to (1) estimate the costs from **rd**i[j] data; (2) evaluate the accuracy of the estimates; and (3) validate the tools and estimation formulas are outlined and explained below. We will conclude with details and results of our first two case studies.

5.2.1 Estimate cost of node executions

1. The cost of one or more executions of some nodes is acquired by calculating the sum of the differences between counts from covering probes taken during a Full instrumentation run. Such data are denoted by **Read**i. This data includes the perturbations.

2. The overhead for certain **Read**i is hypothesized to be approximated by the sum of the differences between adjacent probe counts taken during a Null instrumentation run. Such difference data are denoted by **Overhead**j.

   Other **Read**i have more complicated hypothesized overheads whose approximations are obtained from "suplementary experiments". The code benchmarked in these supplementary experiments is selected to have small total cost relative to the other code under test.

3. Appropriate differences of **Read**i and **Overhead**j values are calculated and can be reported as our estimates for the costs associated with a set of node executions.

5.2.2 Evaluate accuracy of cost estimates

In our preliminary and reported experiments we observed that the related instrumentation overhead could differ from one location in the code under test to another and the amount of code under test could affect this overhead. We approached these problems as follows: The overhead should be measured in the exact same control structure as the code under test, the overhead should be measured for as many probes as will be used in the Full instrumentation run, and the overhead should be measured with the code under test present.

Our Null instrumentation strategy (of displacing the code under test from between the probes but still executing it during the run) was invented to help establish an experimental environment that supports the following:

Hypothesis: Direct and indirect perturbation causes have "approximately equal" effects on the node costs affecting Null, Full and supplementary instrumentation measurements.

In other words, by creating a similar environment for Full and Null instrumentation and limiting the relative magnitude of total supplementary costs we hope that most of the overhead of Full instrumentation will be accounted for.

The accuracy of this hypothesis, i.e., quantifying bounds to clarify the qualitative concept of “approximately equal”, is evaluated as follows:

1. The sum of the all the above estimated (approximately overhead free) costs is reported as the quantity we call **FullResult**.

2. The Raw instrumentation program is run. Raw instrumentation runs exactly two probes, one before all the code under test and the other after. The difference between the two probe values is reported as the quantity we call **RawRead**.

3. The Null Raw instrumentation program is run. Its **Overhead** is reported. The difference quantity **RawResult=RawRead-Overhead** is calculated.

4. The quantities **FullResult** and **RawResult** are compared. Their difference is hypothesized to be a measure of the perturbation effect differences that limit the accuracy of our estimate.

   $$\textbf{Absolute Error} = \textbf{AE} = |(\textbf{FullResult} - \textbf{RawResult})|$$

   $$\textbf{Relative Error} = \textbf{RE} = \frac{\textbf{Absolute Error}}{\textbf{RawResult}}$$

5.2.3 Validate tools and estimation formulas

1. The estimation of costs and the estimates of accuracy evaluation is performed for the instruction count (IC) metric (called “instructions retired” in appendix A of [7]).

2. The IC for each code under test node and each probe node, for Full, Null, Raw and supplementary instrumentations is determined from disassembly listings produced by the Gnu binutils `objdump` utility or the assembly language listing from `gcc`.

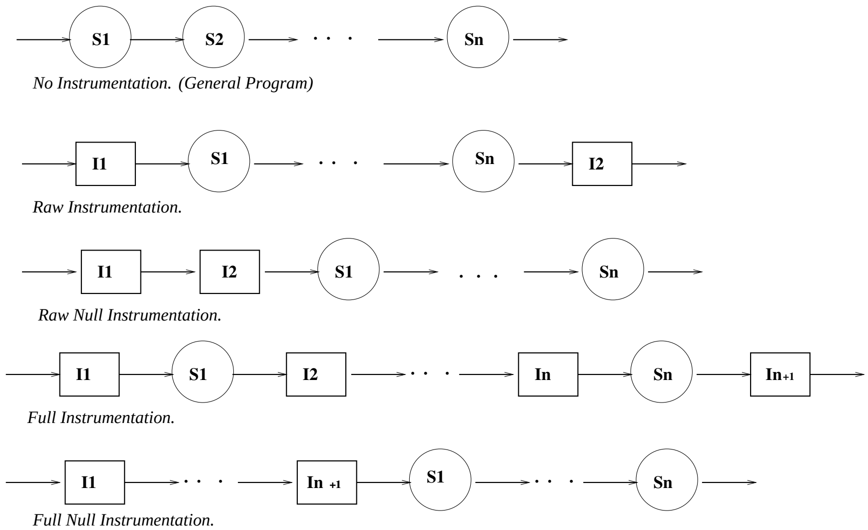


Figure 1: Instrumentation alternatives.

3. The estimated instruction counts are checked against the count predictable from the machine code analysis and known number of times each node is executed. This validates the tool (for instruction count). Approximation choices made in the supplementary experiment design effect explainable differences in the IC checks.

4. The accuracy evaluation (RE in our case study and Table 1) is an indication of the magnitude of all the remaining perturbations including compiler produced (instruction level) indirect perturbations that detract from equality in the main hypothesis. The explainable differences (item 3 above) in the IC checks account for non-zero relative error in IC.

5.3 Case Study: For loop

For straight line code our perturbation model predicts the true performance from instrumented execution with a very small relative error. For codes with loops there are control

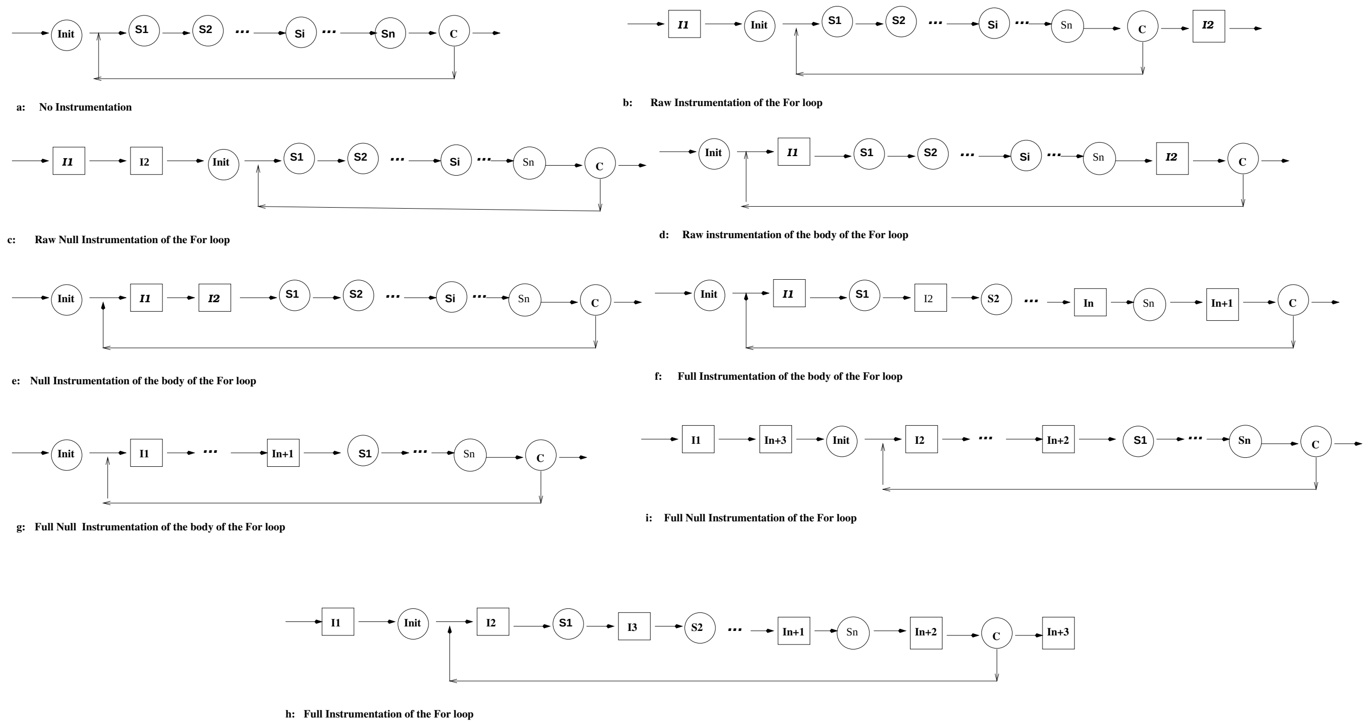


Figure 2: Instrumentation alternatives for `for` loops.

operations for which the perturbation model is not as simple. Inside the body of a `for` loop, the instructions executed between bottom probe and the next iteration's top probe are subject to this type of perturbation. Instrumentation alternatives for `for` loops can be seen in figure 2. Figures 3 and 4 show model extensions required for constant bound `for` loops under full compiler optimization: The first iteration is unconditional and the increment test is moved to the end. The data for this model is generated from 8 different experiments. Figure 3 details the Read and Overhead definitions applied to Full, FullNull, Raw and RawNull instrumentations of our `for` loop. In an ideal perturbation free environment we would expect **FullResult==RawResult**. In the real environment these two values differ.

It may seem that **Overhead4** is the appropriate overhead to subtract from **Read1** to estimate the cost of `for` loop initialization. However, it is not appropriate because **Read1** also includes any indirect perturbations from the compiler generated instructions from the `j++` operation in the second probe **rd1**[`j++`] before this probe's RDPMC is actually executed.

To estimate the overhead correction appropriate for **Read1**, we did the supplementary experiment to estimate the costs of one `for` loop initialization (figure 4c to d). We estimate this cost by Z1=**Read8-Overhead8**. Secondly we report the estimated overhead on `for` loop initialization by **Overhead1**=**Read1**-Z1. In fact the instruction count result for **Overhead1** differed from **Overhead4** when the compiler generated probe code was used; **Overhead1** became equal to **Overhead4** when we used hand optimized probe code.

Supplementary experiments (figure 4a and 4d) are needed for estimating **Overhead2** and **Overhead3** for similar reasons. There are **Overhead2** and a few control operation costs between **rd1**$[(M \star N) - 1]$ and **rd0**[1], and **Overhead3** and a few control operation costs between **rd1**$[M \star j]$ and **rd1**$[(M \star j) - 1]$ for each $j > 0$. Note that loop control operations belong to the code under test. To estimate **Overhead2** and **Overhead3** subtract the cost estimate for control operations from **Read2** and **Read3**. In figure 4, Y=**Read7 - Overhead7** is the cost estimate for loop operations when $N > 1$ and Y1=**Read7.1-Overhead7.1** is the cost estimate for loop operations when $N == 1$. Y and Y1 each include the control operations for one or more iterations plus a few additional loop related operations. One of these additional operations is the initialization. Having computed the cost for initialization (Z1), we estimate the cost for control operations executed between **rd1**$[(N \star M) - 1]$ and **rd0**[1] by $W1 = Y1 - Z1$ and the cost for control operations executed between **rd1**$[M \star j]$ and **rd1**$[(M \star j) - 1]$ for each $j > 0$ by $W = Y - Z1$. W and W1 still include the cost for small loop related operations. Since estimating the cost for these operations will introduce new unaccounted operations, and their impact on the relative error is not much when N is large, we decided to approximate them by zero. This is an example of an experimental approximation choice whose effect is observed and explained under step (3) of section 5.2.3.

The arithmetic fact **FullRead == Read1 + Read2 + Read3 + Read5** makes direct use of **Read5** unnecessary[3]. However, we need to estimate the overhead correction appropriate for **Read5**. The Null instrumentation experiment (figure 3b) estimates **Overhead5**. From this experiment **Overhead5** $= \sum_{j=1}^{N}(\mathbf{rd1}[(M \star j) - 1] - \mathbf{rd1}[M \star (j-1)])$. At this point we have estimated all overhead corrections appropriate for **FullRead**. We estimate the net performance of Full instrumentation as FullResult:

FullResult = **FullRead** - (**Overhead1** + **Overhead2** + **Overhead3** + **Overhead5**).

From experiments in figure 3c and 3d: **RawResult** = **RawRead** - **Overhead6**.

Table 1 lists some of the Pentium Pro performance metrics and the comparison between the relative errors for `blur` and the simple `for` loop for two different types of probe (hand optimized and compiler generated). As can be seen from this table, using hand optimized probes results in a considerable reduction in the relative error of most of the performance metrics. The hand optimized probes result in fewer and more efficient instructions than do the compiler generated probes. The decreased relative errors shown in Table 1 for the more costly **blur** code compared to the simple **for** loop confirms that the unaccounted perturbations mentioned before tend to be constant.

Figure 5 demonstrates all the ideas explained so far for Full instrumentation of a simple `for` loop in a more detailed picture. On the left is the C source above the source execution

[3] We did acquire Read5 and verified this.

model of a `for` loop (without optimization). On the right side is our control flow model for the optimized assembly code generated by gcc.

5.4 Numerical Example and Results:

We use the following example to demonstrate our model. This example is using optimized assembly language probe code on the simple `for` loop in figure 5 with instruction count as the performance metric under study. Data in this example are read from the performance counters, but they also can be obtained by looking at the assembly code generated by the compiler.

Let N = 100, Let M = 2. **Read1** = 10, **Read2** = 990, **Read3** = 10, **Read7** = 209, **Overhead7** = 7,

FullRead = 1810, **Overhead5** = 700, **RawRead** = 310 , **Overhead6** = 7

Read7.1 = 12 , **Overhead7.1** = 7, **Read8** = 8 , **Overhead8** = 7

Z1 = **Read8** - **Overhead8** = 8 - 7 = 1

Y = **Read7** - **Overhead7** = 209 - 7 = 202

Y1 = **Read7.1** - **Overhead7.1** = 12 - 7 = 5

W = Y - Z1 = 202 - 1 = 201

W1 = Y1 - Z1 = 5 - 1 = 4

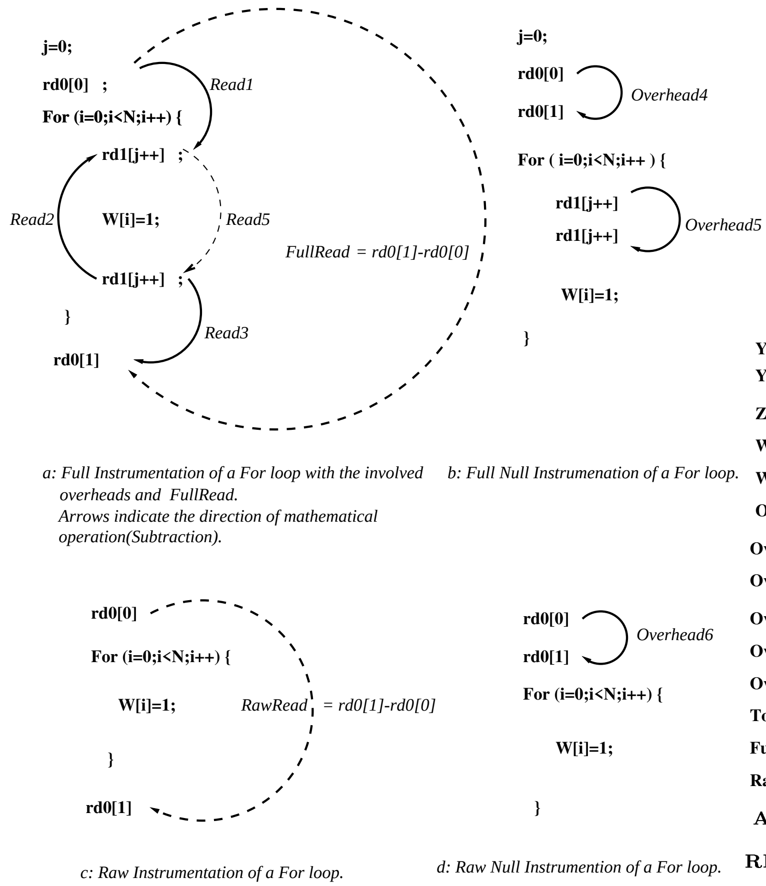
Overhead1 = **Read1** - Z1 = 10 - 1 = 9

Overhead2 = **Read2** - W = 990 - 201 = 789

Overhead3 = **Read3** - W1 = 10 - 4 = 6

FullResult = **FullRead** - (**Overhead1** + **Overhead2** + **Overhead3** + **Overhead5**)

FullResult = 1810 - (9 + 789 + 6 + 700) = 306



Let N = Number of iterations.

Let M = Number of probes inside the loop.

Read1 = rd1[0] - rd0[0]

$$\mathbf{Read2} = \sum_{j=1}^{N-1} (rd1[M \star j] - rd1[(M \star j) - 1])$$

Read3 = rd0[1] - rd1[(N $\star$ M) - 1]

Overhead7 = rd0[1] - rd0[0] **In Empty.For Null**

Read7 = rd0[1] - rd0[0] **In Empty.For Raw**

Overhead7.1 = rd0[1] - rd0[0] **In Empty.For Null , N==1**

Read7.1 = rd0[1] - rd0[0] **In Empty.For Raw , N==1**

Overhead8 = rd0[1] - rd0[0] **In Init.For Null**

Read8 = rd0[1] - rd0[0] **In Init.For Raw**

Y = Read7 - Overhead7

Y1 = Read7.1 - Overhead7.1

Z1 = Read8 - Overhead8

W = Y - Z1

W1 = Y1 - Z1

Overhead1 = Read1 - Z1

Overhead2 = Read2 - W

Overhead3 = Read3 - W1

Overhead4 = rd0[1] - rd0[0] **In Full Null**

$$\mathbf{Overhead5} = \sum_{j=1}^{N} (rd1[(M \star j) - 1] - rd1[M \star (j - 1)])$$

Overhead6 = rd0[1] - rd0[0] **In Raw Null**

TotalOverhead = Overhead1+Overhead2+Overhead3+Overhead5

FullResult = FullRead - TotalOverhead

RawResult = RawRead - Overhead6

$$\mathbf{AE} = |(FullResult - RawResult)|$$

$$\mathbf{RE} = |\frac{FullResult - RawResult}{RawResult}|$$

Figure 3: Experiments for capturing performance counts of a `for` loop.

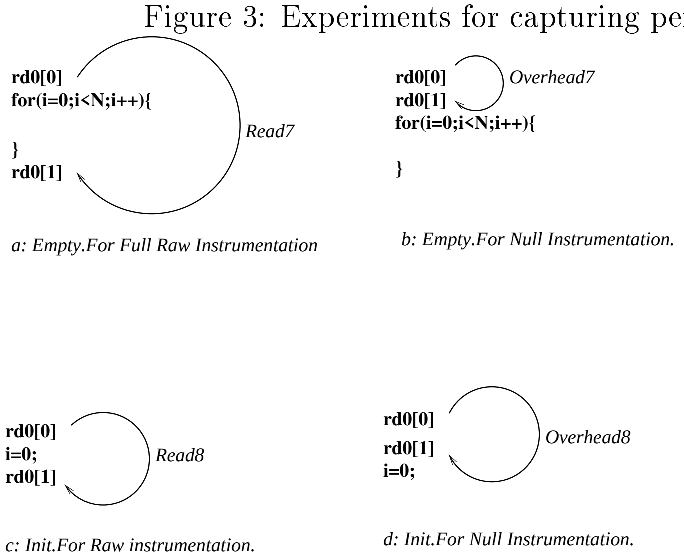


Figure 4: Supplementary experiments for capturing performance counts of a `for` loop.

$\mathbf{RawResult} = \mathbf{RawRead} - \mathbf{Overhead6} = 310 - 7 = 303$

$\mathbf{AE} = |(\mathbf{FullResult} - \mathbf{RawResult})| = 3$

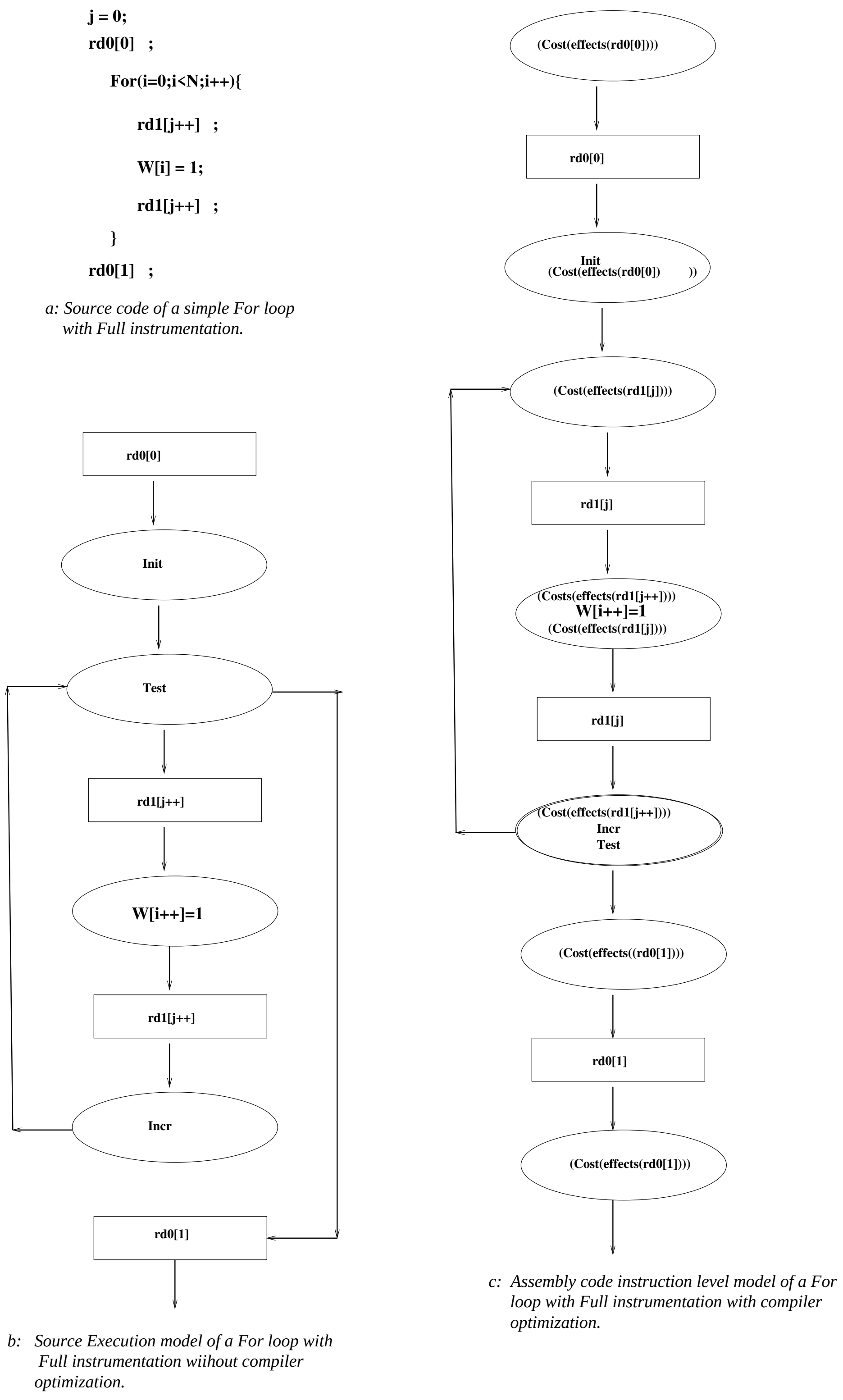
$\mathbf{RE} = |\frac{\mathbf{FullResult - RawResult}}{\mathbf{RawResult}}| = 0.0099$

ID	Performance metric	Mnemonic Event name	Blur Code RE	Blur Code RE(Optimized)	Simple For loop RE	Simple For loop RE(Optimized)
0	Data references	DATA_MEM_REFS	0.1755	0.0083	0.4959	2.059
1	L1 Dcache misses	DCU_LINES_IN	0.0711	0.1153	1.666	1.4
8	Istalls cycles	IFU_MEM_STALL	0.4520	0.3443	8.2258	4.7
50	Instructions retired (IC)	INST_RETIRED	0.0748	0.0002	0.7711	0.0099
51	UOPS	UOPS_RETIRED	0.1034	0.0001	1.133	0.7444
52	Speculative execution factor	INST_DECODED	0.0824	0.0004	1.7370	0.2578
56	Branch frequency	BR_INST_RETIRED	0.0023	0	2	0
57	Branch mispredict ratio	BR_MISS_PRED_RETIRED	0.0492	0	2	0
58	Branch taken ratio	BR_TAKEN_RETIRED	0.0028	0	2	0
60	Branch Speculation factor	BR_INST_DECODED	0.0023	0.0004	2.0566	0.0485
61	BTB miss ratio	BTB_MISSES	0.0009	0.0002	2	0.5
64	Resource stalls	RESOURCE_STALLS	0.0521	0.0339	0.6500	14.4333
67	Cycles	CPU_CLK_UNHALTED	0.0084	0.0039	0.8050	1.9774

Table 1: Relative Errors for some of the monitored Pentium II processor counter based performance metrics. "Optimized" only indicates hand optimized probe design versus compiler generated probes: compiler optimization (-O2) was specified for code under test.

6 Related Work

We survey some related studies of profiling and instrumentation perturbation analysis. [22, 21, 20] derive a perturbation model based on experimental data that can approximate true performance from instrumented execution in sequential, vector, concurrent and vector-concurrent modes. Our work concentrates on analyzing the source level instrumentation and extends Malony, Reed and Wijshoff's models in [22] to estimate the perturbation overhead



a: Source code of a simple For loop with Full instrumentation.

b: Source Execution model of a For loop with Full instrumentation wiihout compiler optimization.

c: Assembly code instruction level model of a For loop with Full instrumentation with compiler optimization.

Figure 5: Detailed Full instrumentation model of a `for` loop.

with using the Intel Pentium II processor and its performance counters. [27] study the elimination of execution perturbation caused by high-level instrumentation of SPMD programs. Rabbit[16] is a performance counter library for Intel processors and Linux which allows programs to measure their own performance. Each Rabbit probe has a lot of overhead because it is a function call. PAPI[14] is a portable source level application program interface to hardware performance counters. We decided to ignore portability for now so we can study accuracy and validity first. [6] compares the performance of three operating systems. It too uses the Intel Pentium processor's event counters for measurements and device drivers to acquire the count from user mode. This study uses null benchmarks with code under test always removed to estimate the overhead involved in instrumentation. [29] study the profiling and tracing of C++ applications that have been written using a parallel programming framework for high performance scientific applications. They also use source code instrumentation to access hardware performance counters. The classic study [4] compares various overall metrics (CPI, IC, instruction frequencies, etc.) for organizationally similar RISC and CISC processors on the SPEC benchmarks. [5] also uses built-in performance counters to compare the performance of Pentium and Pentium Pro processors. [5] uses the performance counters to show that the Pentium Pro achieves significantly lower CPI than the original Pentium. [1] use flow and context sensitivity in data flow analysis for dynamic program analysis and program profiling. Interesting data structures are used to associate hardware metric totals with every path through a program's call and flow graph. [2] describes the digital continuous profiling infrastructure, a sampling-based profiling system designed to run continuously on production systems. Their sampling relies on the Alpha processor's performance counter hardware.

7 Conclusion and Future Work

Our work concentrates on a microscopic instrumentation of source code statements. Fine grain hardware system behavior on code selected from applications is easy to observe without the interference from operating system activity by loading the benchmark into a dedicated system as a kernel module and activating it as a device driver. Throughout our studies we have found that probe overhead is more accurate taken the code under test is displaced from between probe pairs instead of being simply deleted, in contrast to the construction of the "Null benchmarks" used by [6] which the code under test is removed during Null benchmarking. We observed that even when hardware behavior on optimized code under test is the experimental subject, the compiler's optimizer should not be allowed to schedule the probe code.

Our future research will include using the flow graph to automate the perturbation analysis process. This research will also allow us to begin a research program to apply exokernel[9] type application execution to mature application software currently available for a commodity operating system, and help develop improved infrastructures for processor affinity scheduling of symmetric multiprocessors[33, 32] and other dynamic prediction and system tuning applications.

It is an open question whether the observed error magnitudes in non-IC metrics can be explained from assumptions about interinstruction effects in our hardware. But we can now at least observe these metrics and their errors.

8 Acknowledgements

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *1997 ACM SIGPLAN conference on Programming Language Design and Implementation, Las Vegas, Nevada*, June 15-18 June, 1997.

[2] Jennifer M. Anderson, Lance M. Berg, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *ACM Transactions on Computer Systems, Vol 15, No. 4, November 1997, pages 357-390*, 1997.

[3] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *In Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems. pages 115-125*, 1990.

[4] Dileep Bhandarkar and Douglas W. Clark. Performance from architecture: Computing a risc and a cisc with similar hardware organization. In *Proceedings of the fourth international conference on architectural support for programming languages and operating systems, pages 310-319*, 1991.

[5] Dileep Bhandarkar and Jason Ding. Performance characterization of the pentium pro processor. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, 1997.

[6] J.Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Selzer, and Michael D. Smith. The measured performance of personal computer operating systems. In *The Proceedings of the 15th ACM Symposium on Operating System Principles.*, 1996.

[7] Intel Corporation. Intel architecture software developer's manual, volume 3: System programming guide. In *Intel Corporation, Order number 243192.*

[8] Intel Corporation. Pentium pro family developer's manual, order number 242692.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles , 1995, Pages 251 - 266*, 1995.

[10] A. J. Goldberg and J. L. Hennessy. Mtool:an integrated system for performance debugging shared memory multiprocessor applications. In *IEEE Transactions on Parallel and Distributed Systems,28-40.*, 1993.

[11] S. Graham, P. Kessler, and M. Mckusick. gprof:a call graph execution profiler. In *SIGPLAN Not. 17, 6(June), 120-126.*, 1982.

[12] http://developer.intel.com/design/perftool/vtune.

[13] or source release ftp://gcc.gnu.org/pub/releases http://gcc.gnu.org/onlinedocs.

[14] http://icl.cs.utk.edu/projects/papi/dodugm99/papi.html.

[15] http://www.gimp.org.

[16] http://www.scl.ameslab.gov/Projects/Rabbit.

[17] http://www.uni paderborn.de/pc2/projects/warp/sproftool/main.htm.

[18] Raj Jain. *The Art of Computer Systems Performance Analysis Techniques for experimental Design, Measurement, simulation, and Modeling.* WILEY, 1992.

[19] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, 1995.

[20] Allen D. Malony. Event-based performance perturbation: a case study. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming , pages 201-212*, 1991.

[21] Allen D. Malony and Daniel A. Reed. Models for performance perturbation analysis. In *Conference proceedings on ACM/ONR workshop on parallel and distributed debugging , Pages 15 - 25*, 1991.

[22] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, 1992.

[23] Digital Unix man page.

[24] David A. Patterson and John L.Hennessy. *The Radon Transform and Some of its Applications.* Morgan Kaufmann Publishers, San Francisco, California, 1996.

[25] J. F. Reiser and J. P. Skudlarek. Program profiling problems, and a solution via machine language rewriting. In *SIGPLAN Not. 29, 1(Jan.),37-45*, 1994.

[26] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The simos approach. In *IEEE Parallel Distrib. Tech. 3, 3(Fall).*, 1995.

[27] Sekhar R. Sarukkai and Allen D. Malony. Perturbation analysis of high level instrumentation for spmd. In *Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming*, 1993.

[28] Eric Schnarr and James R. Larus. Instruction scheduling and executable editing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS'96), Tucson, Arizona*, 1996.

[29] Sameer Shende, Allen D. Malony, Janice Cuny, Kathleen Lindlan, Peter Beckman, and Steve Karmesin. Portable profiling and tracing for parallel, scientific application using c++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pages 134-145, August 1998*, 1998.

[30] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 1998 (version 2.8.1).

[31] MIPS Computer Systems, Sunyvale, and Calif. Umips-v reference manual (pixie and pixstats). 1990.

[32] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Proceedings of the 8th international conference on Architectural support for programming languages and operating systems.*, pages 127–138, 1998.

[33] Yong Yan, Canming Jin, and Xiaodong Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1):70–81, 1997.

[34] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r20000 performance counters. In *In Proceedings of Supercomputing.*, 1996.

[35] X. Zhanga, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. Operating system support for automated profiling and optimization. In *In Proceedings of the 16th ACM Symposium on Operating Systems Principles.*, 1997.