PDMA Toolkit[1]

Aditya Kalyanpur Vikas Raykar Sadagopan Srinivasan

Department of Computer Science, Department of Electrical Engineering,

University of Maryland, College Park

ABSTRACT

In this report, we present PDMA – **Papi Dyninst Memory Analysis** Toolkit that can be used to test programs for memory bottlenecks such as cache misses. The tool uses underlying hardware counters supported by existing machines to monitor memory-specific events and patches the runtime code of the program in order to monitor these events. The performance analysis results are displayed using histograms and stacked bar charts that are hyperlinked with the original source code. We use our tool to conduct tests on specific benchmarks such as *Parkbench* and *FFT* that successfully demonstrate its application utility. We present the results of these tests, highlight key factors and implications, and suggest future plans for research.

1. Introduction

Most of the modern computer systems have deep memory hierarchies. As a result, very often, significant performance improvements can be obtained by restructuring the code either in terms of the data layout or the way the program is written. Hence a tool, which can pinpoint the memory bottlenecks accurately in terms of lines in the source code, would be very beneficial for the programmer. Once the memory bottleneck has been identified, the programmer can make use of the knowledge of the memory hierarchy to restructure the code.

There are two ways in which a memory bottleneck can be reported. The memory bottleneck can be associated either with the data or the source code or possibly both. In data based tools, typically, some metrics are reported for different data structures (like arrays) in the source code. In source based method, bottlenecks are reported in terms of actual lines in the source code. We use the latter methodology in our PDMA toolkit.

The first step in designing such a tool is to identify the sources of memory bottlenecks and develop a taxonomy of memory loss performance metrics. Most modern microprocessors have two levels of cache (L1 and L2 cache), although some of them now have three levels. The latency of data access becomes greater with each level of memory hierarchy. For example, a typical L1 cache hit could take 2 to 3 cycles whereas a L1 cache miss satisfied by L2 cache hit takes around 8 to 10 cycles. A L2 cache miss satisfied from the main memory with no TLB miss takes around 75 to 250 cycles and a TLB miss requiring the reload of TLB takes around 2000 cycles. In our tool the basic metrics are L1 Instruction cache miss, L1 Data cache miss (load and store misses). This is because in modern microprocessors most of the cache accesses (almost 90%) are served by the L1 cache and it is the L1 miss penalty, which constitutes a major overhead in a lot of applications. Additionally, our tool relates these three metrics to the program execution time to give a measure of the significant time lost in each memory bottleneck.

Thus, to summarize, our goal is to isolate and emphasize the effect of cache misses in the program. For this, our tool needs to perform the following two tasks:

- Report the various L1 misses viz., instruction, load and store misses occurring in each line of the source code. A histogram is generated which shows the percentage of the total cache misses caused by that particular line. This helps in identifying the *Hotspots* or memory bottlenecks in the source. We also wanted to build a GUI which could hyperlink the histogram to the relevant source code fragment.
- The second goal was to give an estimate of the fraction of the total time spent at a line due to each memory bottleneck. We used a stacked bar chart to accomplish this as explained in detail in section 3.2.

In order to achieve these goals, one approach is to run the program using a memory simulator. However simulators are prohibitively slow to be used as a tool to detect memory bottlenecks and the accuracy of the performance analysis results are constrained by the accuracy of the simulator. The other popular method is software based instrumentation. This method, if not properly implemented, can inevitably perturb the execution of the program.

MTool is one such tool which uses low overhead timers and basic block counting. It augments a program with specific instrumentation, which perturbs the program's execution as little as possible while generating enough information to isolate memory and synchronization bottlenecks. It estimates the ideal execution time for a section of code by generating a CFG of that code, heuristically determines the number of instructions issued by minimally counting path traversals along the CFG, and appropriately scales this count using processor-specific information such as average execution time per instruction. It then compares this ideal execution value with the actual measured value to evaluate bottlenecks.

However, this technique suffers from two main drawbacks:

1) The measured execution times are not accurate since they use system-level software commands to obtain them.

[1] This report was written as part of the course project for CMSC 714: High Performance Computing Fall 2003 offered by Dr. Jeff Hollingsworth

2) The heuristically determined ideal execution times are not precise, as constrained by their methodology.

Nowadays most processors have a small number of performance-dedicated special purpose registers called **Hardware Performance counters**. This special set of registers count events, which are occurrences of specific signals and states related to the processors' architecture. For example Pentium IV has as many as eighteen counters, AMD Athlon has four and UltraSparc II has two. Some processors have more sophisticated hardware for recording data such as data and instruction addresses for an event, and pipeline or memory latencies for an instruction. **We use these counters to count the number of cache misses (instruction and data) that occur in a given program**. Our approach uses **PAPI** [1] to access the hardware counters, thereby reporting more accurate execution times.

Furthermore, we use **Dyninst** [2] to dynamically insert this PAPI code into the running binary, thereby allowing any section of the code to be monitored on the fly. Dynamically instrumenting the binary avoids parsing and recompilation of the source code thereby avoiding interference of the instrumentation with various compiler optimizations. Moreover this avoids the hassles of statically adding code to each and every benchmark.

There are two ways of using hardware performance counters. One is the counting mode to collect aggregate counts of event occurrences and the second is the statistical sampling mode to collect profiling data based on counter overflows. In order to generate the histograms we use the latter approach, which works as follows: when an event exceeds a threshold, an interrupt occurs and a signal is delivered with a certain number of arguments. Among these arguments is the interrupted thread's stack pointer and register set. The register set contains the program counter, the address at which the process was interrupted when the signal was delivered. Our tool extracts this address and hashes the value into a specific file. Upon program completion, the result files are associated with symbolic information contained in the executable. So we have a line-by-line account of where the counter overflow occurred in the program.

The rest of the report is organized as follows: Section 2 briefly describes the above two API's we use i.e. PAPI and Dyninst (while noting our usage experience); Section 3 explains how they are integrated into the overall architecture and working of the toolkit; Section 4 reports results on some tested benchmarks; Section 5 covers some important issues that need to be addressed with regard to the enhancement of our toolkit; and Section 6 finally concludes.

2. Components of PDMA: Architecture and Working

2.1 PAPI

2.1.1 Overview

The Performance API (PAPI) specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors.

2.1.2 Usage Experience

- Initially we tried to setup our entire tool on Linux. However, installing PAPI on a Linux box requires the kernel to be patched. Furthermore, patches are not available for all distributions of Linux. We only managed to install it on a single isolated machine with Red Hat Linux 8.0 and this proved cumbersome. So we decided to work on the Solaris platform instead and installed it on *tau.umiacs.umd.edu* where all of us could have access.
- PAPI provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. **We used the low level API for our tool**.
- PAPI has the concept of Preset events, also known as pre-defined events which are a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide performance counters and give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Not all events can be measured on all platforms. We wrote a program which prints out the hardware information and the available events. For our tool we are mainly interested in the following events which are available on UltraSPARC III.
    - **PAPI_L1_ICM**: Level 1 instruction cache misses
    - **PAPI_L1_LDM**: Level 1 load misses
    - **PAPI_L1_STM**: Level 1 store misses
    - **PAPI_TOT_CYC**: Total Cycles
    - **PAPI_TOT_INS**: Instructions completed
- The number of hardware events that can be monitored simultaneously is limited by the number of hardware counters, which in turn depends on the processor. The Sun Ultra Sparc processor which we worked on had two hardware counters. As a result we monitored two events simultaneously. However, PAPI also supports multiple events monitoring with multiplexed counters, but the results are not as accurate.
- PAPI 2 provides the ability to call user-defined handlers when an overflow occurs, which is accomplished by setting up a high-resolution interval timer and installing a timer interrupt handler. For the systems that do not support counter overflow at the operating system level, PAPI uses the signal, SIGPROF, by comparing the current counter value against the threshold. If the current value exceeds the threshold, then the user's handler is called from within the signal context with some additional arguments. These arguments allow the user to determine which event overflowed, how much it overflowed, and at what location in the source code. More specifically there is low level API which returns the address where an overflow occurred.
- PAPI 2 can support only one event for the overflow mechanism. However the latest release PAPI 3-beta version can support multiple event monitoring. However the function *PAPI_get_overflow_address()*

is not available in PAPI 3. Instead there is a statistical profiling API - *PAPI_profil* which builds the histogram. A PC histogram can be generated on any countable event.

- On most systems, overflow is emulated in software by PAPI. Only on the UltraSparc III and IRIX does the operating system support true interrupt on overflow. The emulation handler in PAPI runs every millisecond, therefore values have to be chosen that will overflow frequently but not too frequently.
- There is no documentation on stopping the profiling functions. However one of their FAQ's says that the user must call the overflow or the profiling function with the handler or buffer set to NULL and the threshold to 0 after having called PAPI stop.

2.2 Dyninst

2.2.1 Overview

The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary. At first thought, this may seem like a bizarre goal, however there are several practical reasons we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance problem, it might be necessary to insert additional instrumentation into the program to understand the problem.

Another application is performance steering; for large simulations, computational scientists often find it advantageous to be able to make modifications to the code and data while the simulation is executing. Dyninst provides the ability to insert the code into a running program.

2.2.2 Usage Experience

- Installing Dyninst on a Linux box requires few library files to be installed including *libdwarf*, which is not available for Linux mandrake. Though the source code was available for *libdwarf*, the compiled code failed for inexplicable reasons. Though the RPM's were available for Red Hat, we had access to only one such machine. Therefore we opted to install it on *tau*, which we all had access to and on which we had successfully installed PAPI beforehand.
- Dyninst provides the ability to insert code snippets at any point in the binary. It also provides the ability to program using Dyninst functions. Given our approach, we found it tedious to use these pre-defined functions to generate the PAPI calls. Instead, we found it easier to create a library for the necessary PAPI functions by compiling our PAPI functions with Dyninst and PAPI libraries.
- We were interested in the following Dyninst functions:
    - *BPatch_function*: This function finds the specific functions in the mutatee where the user wants to insert the code snippets.
    - *BPatch_findPoint*: This function finds the exact point (entry/exit) in the function where the code is inserted.
    - *insertSnippet*: This function inserts the code snippet in the mutatee at the specified point.
- The other major feature of Dyninst which we were interested in but couldn't make use of was *bpatch_thread!->isTerminated()*. This function is used to check the termination status of the mutatee. We could have potentially used *waitForStatusChange()* (which waits until there is a status change to some thread that has not yet been reported by either *isStopped()* or *isTerminated()* and returns true) along with the above function but for the following problem: the mutatee hung every time this feature was used. A possible explanation for this behavior was a deadlock between the PAPI library calls and this specific feature of Dyninst. When a normal code snippet (not containing any PAPI calls) was inserted the mutatee worked fine. It would be helpful if the thread library calls by Dyninst had more documentation.
- We also faced an interesting scenario wherein the mutator caused a segmentation fault whenever it found a function in a specific benchmark. We couldn't identify the actual reason behind this. We believe that this was due to an incorrect compilation procedure or something invalid in the benchmark code.
- When using the *addr2line* command (which translates an address associated with an executable to the corresponding source code line), we found that the translated addresses were those of system library files and not the actual source code. This problem was fixed by precluding the following libraries, *"-lterm -lcurses"* from being linked in the compilation procedure of the mutatee.

3. Integration of PAPI/Dyninst into the PDMA Model

A high-level architectural view of our system is presented in **Figure 1.** The program *PAPIProbe.c* uses the PAPI API to monitor user-specified memory related bottlenecks. Dyninst is used to patch the runtime process of the test application with *PAPIProbe.c*. Currently, the probe is inserted at the start of execution, and hence the entire program trace is monitored. However, we can use the Dyninst functionality to insert the probe into any section of the code as determined by the user. The probe is used to access **special-purpose hardware counters** associated with each memory bottleneck, and when the counter value exceeds a variable **threshold** parameter, the probe dumps the memory address that caused the bottleneck to a specific file, maintaining a separate count for each address. Thus for each memory bottleneck, a separate file is dumped that stores a list of all locations (addresses) that were affected as a result of the bottleneck along with the number of times it was affected. The post-processing scripts (as explained in the next section) then act on the dumped files in order to reformat them for the GUI.

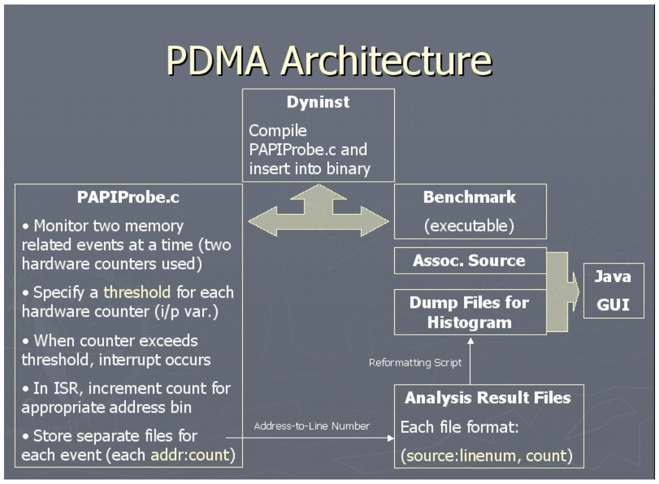

PDMA Architecture
Dyninst
Compile PAPIProbe.c and insert into binary
PAPIProbe.c
• Monitor two memory related events at a time (two hardware counters used)
• Specify a threshold for each hardware counter (i/p var.)
• When counter exceeds threshold, interrupt occurs
• In ISR, increment count for appropriate address bin
• Store separate files for each event (each addr:count)
Benchmark
(executable)
Assoc. Source
Dump Files for Histogram
Java GUI
Reformatting Script
Analysis Result Files
Each file format:
(source:linenum, count)
Address-to-Line Number

Figure 1

3.1 Post-Processing

The post-processing stage consists of two script files, which take the output dumped by PAPIProbe and reformat them to make it easier for the GUI to analyze and display. The first script (*Address-to-Line Number*) converts each memory address location (present in the output files dumped by PAPIProbe) into the corresponding source code and line number that accessed it (see **Figure 1**). The second script parses the output files generated by the first script, removes redundancies, calculates relative percentages from the absolute counts and dumps a final set of files for the GUI.

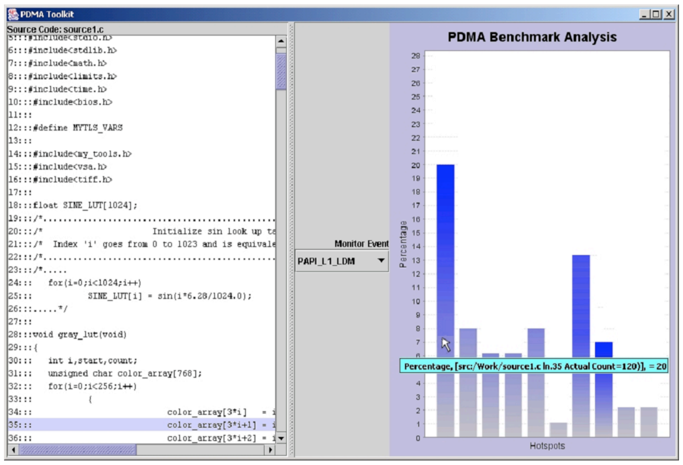
3.2 GUI

3.2.1 Design

The purpose of the graphical user interface is to intuitively display the performance analysis results obtained in our pre-processing stage by PAPI and Dyninst. In coming up with design goals of our GUI, we studied related research tools (MTool, SvPablo) and arrived at the following essential set of features that our GUI needed to support:

Task #1: Allow the user to monitor a specific type of memory bottleneck (say L1 Cache Miss) and observe its impact in the program intuitively

Solution: The GUI has a drop-down list box from where the user selects a specific memory-related event to monitor, and a corresponding histogram is shown highlighting all the hotspots in the program for that event. Each bar in the histogram corresponds to a single hotspot (line of code) in the program and displays the occurrence of the bottleneck at that line in terms of actual count and relative percentage of the whole (see **Figure 2**).



```
5:::#include<stdio.h>
6:::#include<stdlib.h>
7:::#include<math.h>
8:::#include<limits.h>
9:::#include<time.h>
10:::#include<bios.h>
11:::
12:::#define MYTLS_VARS
13:::
14:::#include<my_tools.h>
15:::#include<vsa.h>
16:::#include<tiff.h>
17:::
18:::float SINE_LUT[1024];
19:::/*........................................
20:::/*                    Initialize sin look up ta
21:::/*  Index 'i' goes from 0 to 1023 and is equivale
22:::/*........................................
23:::/*.....
24:::   for(i=0;i<1024;i++)
25:::           SINE_LUT[i] = sin(i*6.28/1024.0);
26:::.....*/
27:::
28:::void gray_lut(void)
29:::{
30:::   int i,start,count;
31:::   unsigned char color_array[768];
32:::   for(i=0;i<256;i++)
33:::           {
34:::                        color_array[3*i]   = i
35:::                        color_array[3*i+1] = i
36:::                        color_array[3*i+2] = i
```

Figure 2

Task #2: Pinpoint exact location of hotspot in terms of actual source code

Solution: Each hotspot (bar) in the histogram is hyperlinked with the corresponding source file and line number, which is highlighted for the user when clicked. This allows the user to quickly navigate between cause of problem and its effect.

Task #3: Determine the actual impact of each memory bottleneck in terms of program execution time

Solution: For a given hotspot, we count the number of processor cycles consumed by each memory-bottleneck event. For our calculations, we assume all L1 (instruction and data) misses to be a L2 hit and use a constant miss penalty of 8 cycles (obtained from the web for the *tau* architecture, which is the SUNW4u sunfire SPARC machine [3]). This is then related to the total number of cycles issued at that hotspot, thereby computing the relative percentage of time wasted by each bottleneck. A stacked-bar chart is used to demonstrate this effectively (see **Figure 3**).

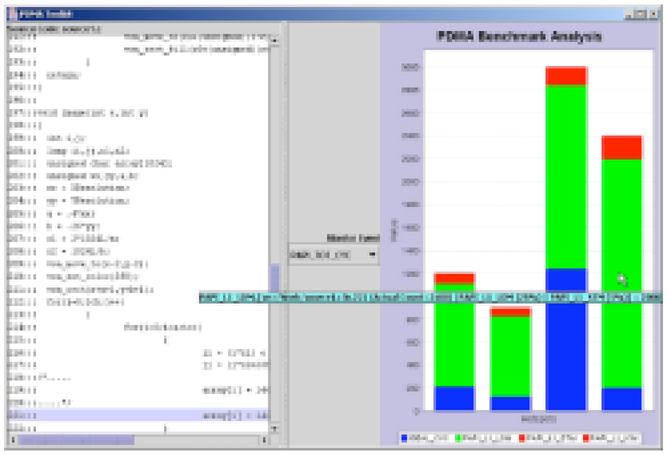


Figure 3

3.2.2 Implementation

We use Java as our programming language to implement the GUI, main reasons being platform independence and the availability of an open-source API to draw charts. The API we use is called **JFreeChart** [4].

In a nutshell, our program titled *PDMA.java* works as follows:

- It accepts the files dumped by the post-processing script
- Parses the data accordingly, storing it in appropriate data structures
- Performs some mathematical calculations aggregating and relating data from all the monitored events, and
- Uses the JFreeChart API to construct the histograms. The API supports different types of charts (line, pie, histogram etc).

The program is invoked by issuing the following at the command line prompt:

java -classpath . PDMA.java PAPI_EVENT1 PAPI_EVENT2 PAPI_EVENT3 PAPI_EVENT4

(where PAPI_EVENTx can be any one of the PAPI monitored events e.g. PAPI_L1_LDM, PAPI_L1_STM etc.). Also note that in most cases, PAPI_EVENT4 is usually PAPI_TOT_CYC (total cycles) since we use it as a reference to display the relative impact of each bottleneck in terms of program execution (see stacked-bar-chart: **Figure 3**)

4. Benchmark Tests

In order to test the PDMA toolkit we ran a number of tests on a few select benchmarks. Initially while beta-testing our tool we wrote a couple of micro-benchmarks to intentionally cause a lot of cache misses (the results for which were reported in our previous presentation [7]). We finally validated our tool against four real benchmarks.

4.1 PARKBENCH [5]

POLY1 and POLY2 from PARKBENCH (PARallel Kernels and BENCHmarks)

The POLY1 and POLY2 benchmarks quantify the dependence of computer performance on memory access bottlenecks. The POLY1 benchmark repeats the polynomial evaluation for each order typically 1000 times for vector lengths up to 10,000, which would normally fit into the cache of a cache-based processor. Except for the first evaluation the data will be found in the cache. POLY1 is therefore an **in-cache** test of the memory bottleneck between the arithmetic registers of the processor and its cache. POLY2, on the other hand, flushes the cache prior to each different order and then performs only one polynomial evaluation, for vector lengths from 10,000 up to 100,000, which would normally exceed the cache size. Data will have to be brought from off-chip memory, and thus POLY2 is an **out-of-cache** test of the memory bottleneck between off-chip memory and the arithmetic registers.

The test results for the POLY benchmarks are summarized in **Table 1**. The table lists all observed hotspots in the source code along with its calculated L1 load cache miss (LDM) count for the POLY1 and POLY2 benchmarks. As can be seen, the number of cache misses in POLY2 is significantly higher than in POLY1 for the same hotspot. Since POLY2 is an out of cache test this behavior is consistent with the benchmark. The resultant histograms for POLY1 and POLY2 L1 load cache misses are displayed in **Figures 4 and 5** respectively. The plots provide for comparative analysis, such as line 65 in the source file *doall.c* causes 23.5% of the total cache misses in POLY1 (about 4000) while 10% of the total cache misses in POLY2 (about 140000). Note that once the user clicks on a particular Hotspot in the histogram, the corresponding line in the source code is highlighted.

Hotspot (Source/Line)	LDM Count (POLY1)	LDM Count (POLY2)
doall.c, line 65	4,000	140,000
doall.c, line 79	2,000	140,000
doall.c, line 93	2,000	140,000
doall.c, line 107	2,000	140,000
doall.c, line 121	1,000	140,000
doall.c, line 135	1,000	140,000
doall.c, line 149	1,000	130,000
doall.c, line 163	1,000	140,000
doall.c, line 177	2,000	140,000
doall.c, line 191	1,000	150,000

Table 1: LDM Hotspots in POLY1 and POLY2

4.2 DSP Benchmarks [6]

We used two DSP benchmarks - **compress** and **FFT**. The above two benchmarks were obtained from the **UTDSP benchmark suite** [6], and are considered important for DSP applications. These benchmarks are not memory bound but computation bound.

- **Compress**: Uses the discrete cosine transform to compress a 128X128 pixel image by a factor of 4 while preserving the information content.
- **FFT**: Performs a Fast Fourier Transform and its inverse. The input data is a polynomial function with pseudo random amplitude and frequency components.

Figure 6 shows the L1 instruction cache misses for the **compress** benchmark (see **Table 2** for actual values). Using this plot the most computationally intensive lines in the source code can be easily identified. **Figure 7** shows the stacked bar chart for the **FFT** bench mark. It displays the amount of cycles lost due to each memory bottleneck at various lines in the source code. For example a lot of L1 instruction cache misses are caused in line 95 and a lot of L1 store cache misses are cause in Line 12.

Hotspot (Source/Line)	Instruction Cache Miss (ICM) Count	Relative Percentage
IO.c ,line 49	1,000	0.6%
IO.c ,line 50	2,000	1.2%
compress.c, line 67	1,000	0.6%
compress.c, line 72	1,000	0.6%
compress.c, line 73	2,000	1.2%
compress.c, line 88	1,000	0.6%
compress.c, line 124	3,000	1.9%

compress.c, line 125	8,000	5.09%
compress.c, line 126	1,000	0.6%
compress.c, line 127	13,000	8.28%
compress.c, line 128	26,000	16.56%
compress.c, line 130	5,000	3.18%
compress.c, line 138	5,000	3.18%
compress.c, line 139	10,000	6.36%
compress.c, line 140	1,000	0.6%
compress.c, line 141	21,000	13.37%
compress.c, line 142	26,000	16.56%
compress.c, line 145	19,000	12.1%
compress.c, line 153	1,000	0.6%
compress.c, line 156	3,000	1.9%
compress.c, line 157	2,000	1.2%
compress.c, line 158	1,000	0.6%
compress.c, line 159	2,000	1.2%
compress.c, line 170	1,000	0.6%
compress.c, line 171	1,000	0.6%

Table 2: **ICM Hotspots in compress**

5. Discussion and Future Work

We would like to enhance our toolkit by addressing the following issues:

- Test the tool on the SPEC [8] benchmarks.
- Currently the GUI is invoked after the results are dumped upon program completion. In the future we would like to dump the results in a shared memory pool, which the GUI can access so that the histogram can be updated while the program runs.
- We would like to explore the features of Dyninst that allow us instrument the code at runtime. Using this feature the user can have control over changing the event to be monitored as the program executes.
- There seems to be a deadlock between the PAPI and Dyninst library calls when the *isTerminated()* command is used. We would like to investigate this.
- We would like to use more than two events at a time in PAPI and check the accuracy of the counters when they are multiplexed.
- We would like to explore the choice of *threshold* more thoroughly by conducting a detailed study on the strategies used to select an optimal threshold value, since it has a direct impact on the accuracy of our results.
- Similar to the threshold parameter, but of less significance, are the multiplication factors used in computing lost cycles in the stacked-bar-chart. We would like to study the implications of choosing specific values for these factors.
- We would like to allow the user to insert the PAPI probe into specific functions/loops of the program and also monitor iterations.
- Finally, we would like to do a comparative study of our model with other existing approaches.

6. Conclusion

In this report, we present our experience in designing and implementing the PDMA toolkit. The primary purpose of this toolkit is to detect the cause and highlight the implications (w.r.t execution time) of specific memory bottlenecks in a given program. We employ *PAPI* to access hardware performance counters to identify such bottlenecks (cache misses), and *Dyninst* to dynamically insert this *PAPI* code into the running binary, thereby allowing any section of the code to be monitored on the fly. The GUI presents the memory bottlenecks as histograms which are linked to the source code. We validate our tool using specific benchmark tests whose outcome is consistent with our understanding of the code. Finally, we list future directions for research in order to build upon the foundations of the framework we've constructed.

7. References

[1] http://icl.cs.utk.edu/papi/index.html

[2] http://www.dyninst.org

[3] http://www.csm.ornl.gov/dunigan/sparc3

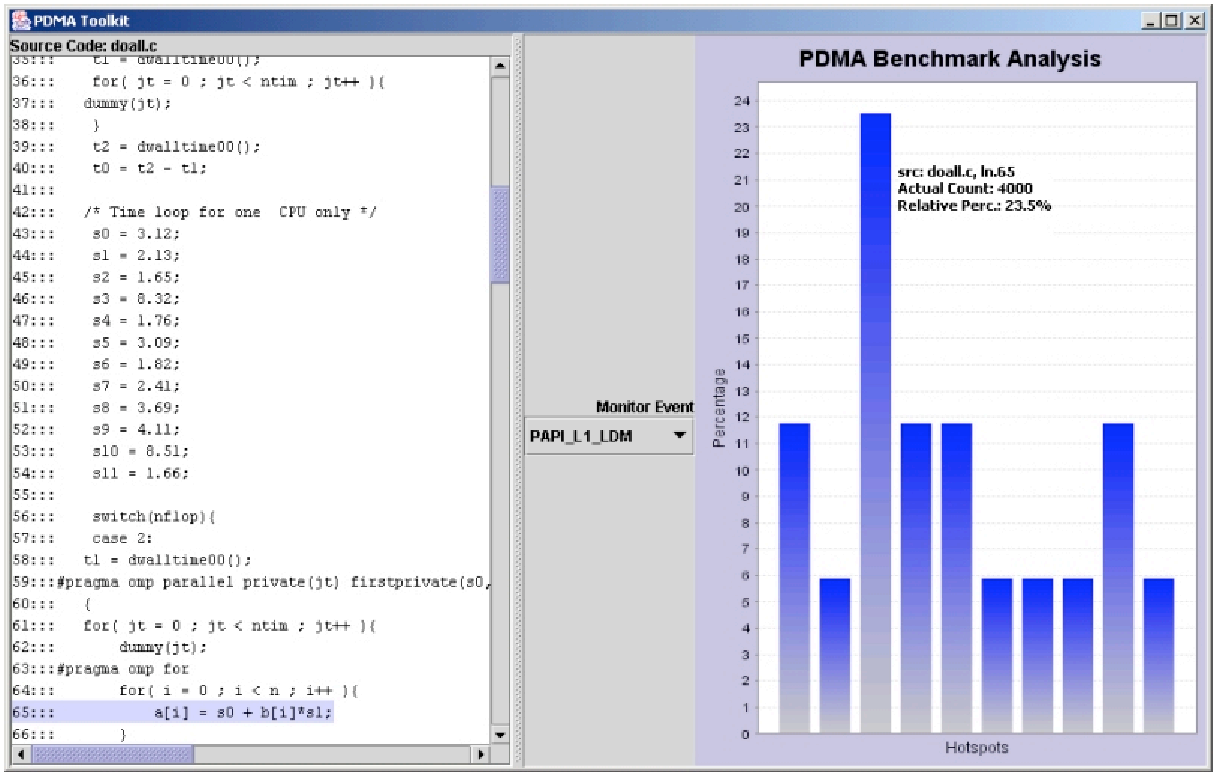
[4] www.jfree.org

[5] http://www.netlib.org/parkbench/

[6] http://www.eecg.toronto.edu/corinna/dsp/infrastructure/utdsp.html

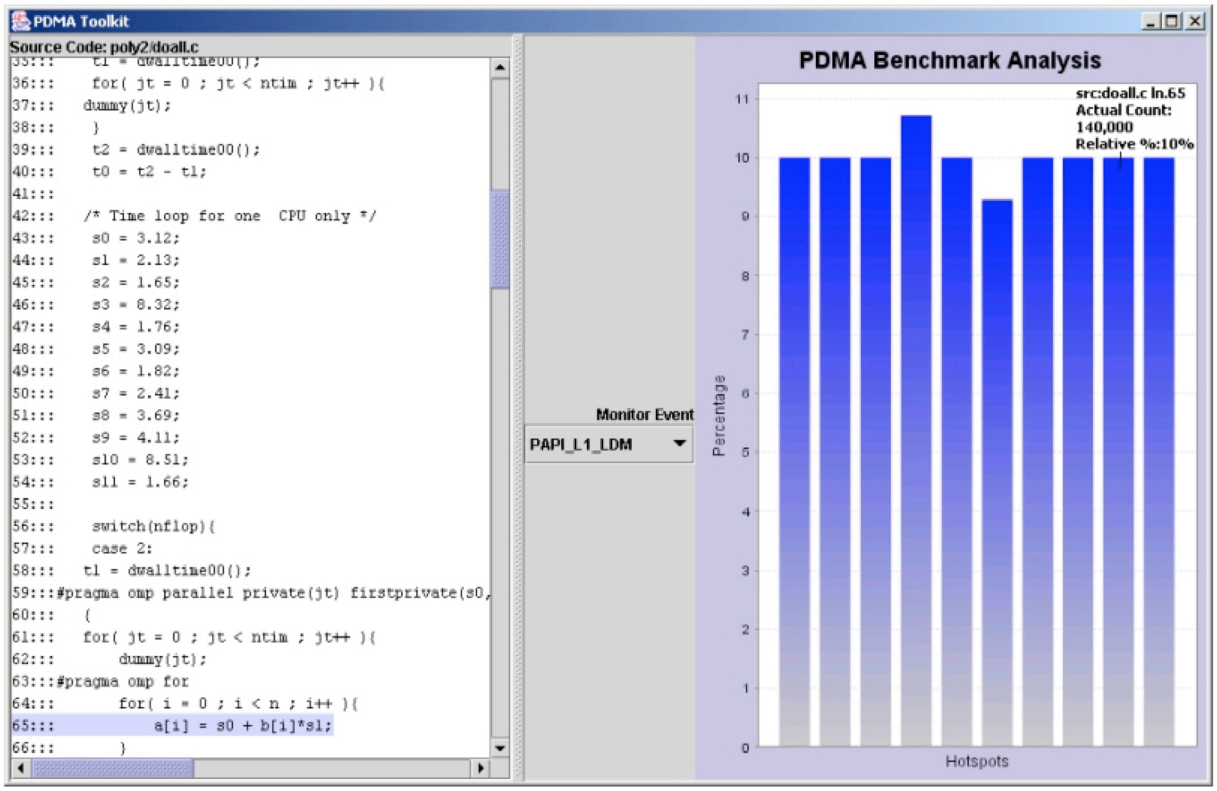
[7] http://www.glue.umd.edu/~sgopan/PDMA.ppt

[8] http://www.specbench.org



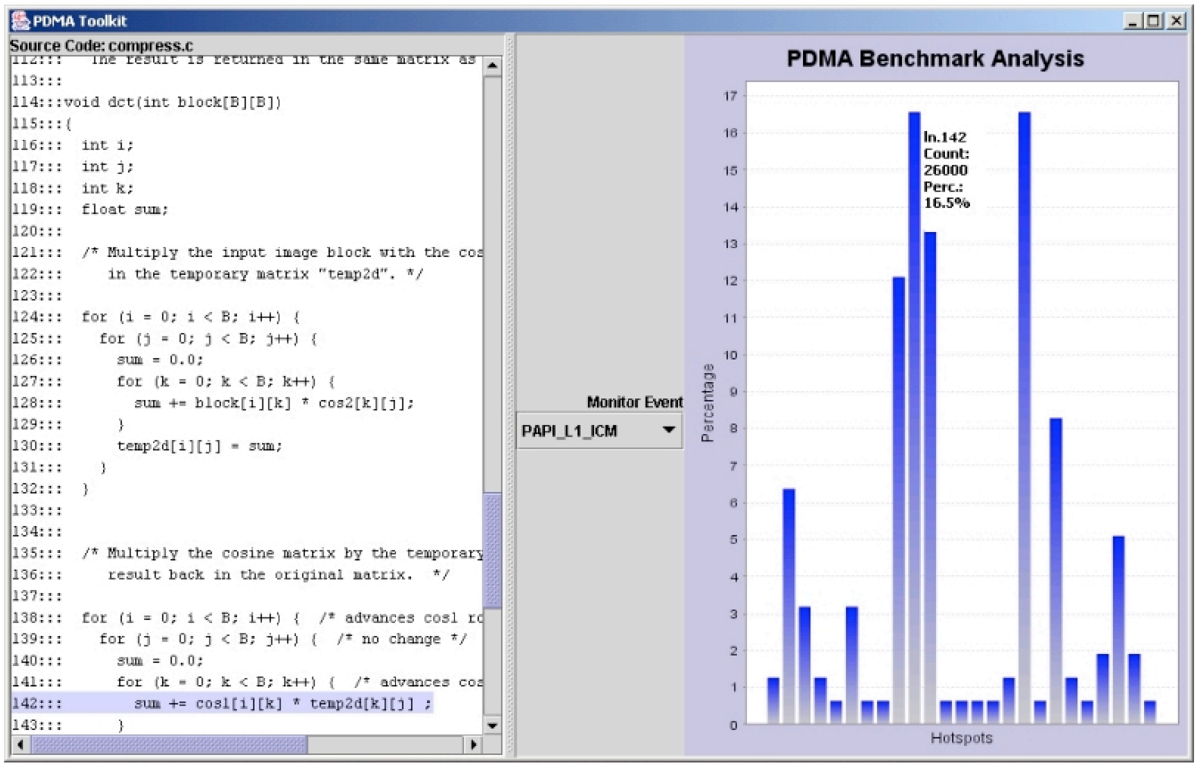
```
35:::    t1 = dwalltime00();
36:::    for( jt = 0 ; jt < ntim ; jt++ ){
37:::   dummy(jt);
38:::    }
39:::    t2 = dwalltime00();
40:::    t0 = t2 - t1;
41:::
42:::   /* Time loop for one  CPU only */
43:::    s0 = 3.12;
44:::    s1 = 2.13;
45:::    s2 = 1.65;
46:::    s3 = 8.32;
47:::    s4 = 1.76;
48:::    s5 = 3.09;
49:::    s6 = 1.82;
50:::    s7 = 2.41;
51:::    s8 = 3.69;
52:::    s9 = 4.11;
53:::    s10 = 8.51;
54:::    s11 = 1.66;
55:::
56:::    switch(nflop){
57:::    case 2:
58:::   t1 = dwalltime00();
59:::#pragma omp parallel private(jt) firstprivate(s0,
60:::   {
61:::   for( jt = 0 ; jt < ntim ; jt++ ){
62:::       dummy(jt);
63:::#pragma omp for
64:::       for( i = 0 ; i < n ; i++ ){
65:::           a[i] = s0 + b[i]*s1;
66:::       }
```

Figure 4. L1 Cache Load Misses for POLY1 Benchmark



```
35:::    t1 = dwalltime00();
36:::    for( jt = 0 ; jt < ntim ; jt++ ){
37:::   dummy(jt);
38:::    }
39:::    t2 = dwalltime00();
40:::    t0 = t2 - t1;
41:::
42:::   /* Time loop for one  CPU only */
43:::    s0 = 3.12;
44:::    s1 = 2.13;
45:::    s2 = 1.65;
46:::    s3 = 8.32;
47:::    s4 = 1.76;
48:::    s5 = 3.09;
49:::    s6 = 1.82;
50:::    s7 = 2.41;
51:::    s8 = 3.69;
52:::    s9 = 4.11;
53:::    s10 = 8.51;
54:::    s11 = 1.66;
55:::
56:::    switch(nflop){
57:::    case 2:
58:::   t1 = dwalltime00();
59:::#pragma omp parallel private(jt) firstprivate(s0,
60:::   {
61:::   for( jt = 0 ; jt < ntim ; jt++ ){
62:::       dummy(jt);
63:::#pragma omp for
64:::       for( i = 0 ; i < n ; i++ ){
65:::           a[i] = s0 + b[i]*s1;
66:::       }
```

Figure 5. L1 Cache Load Misses for POLY2 Benchmark

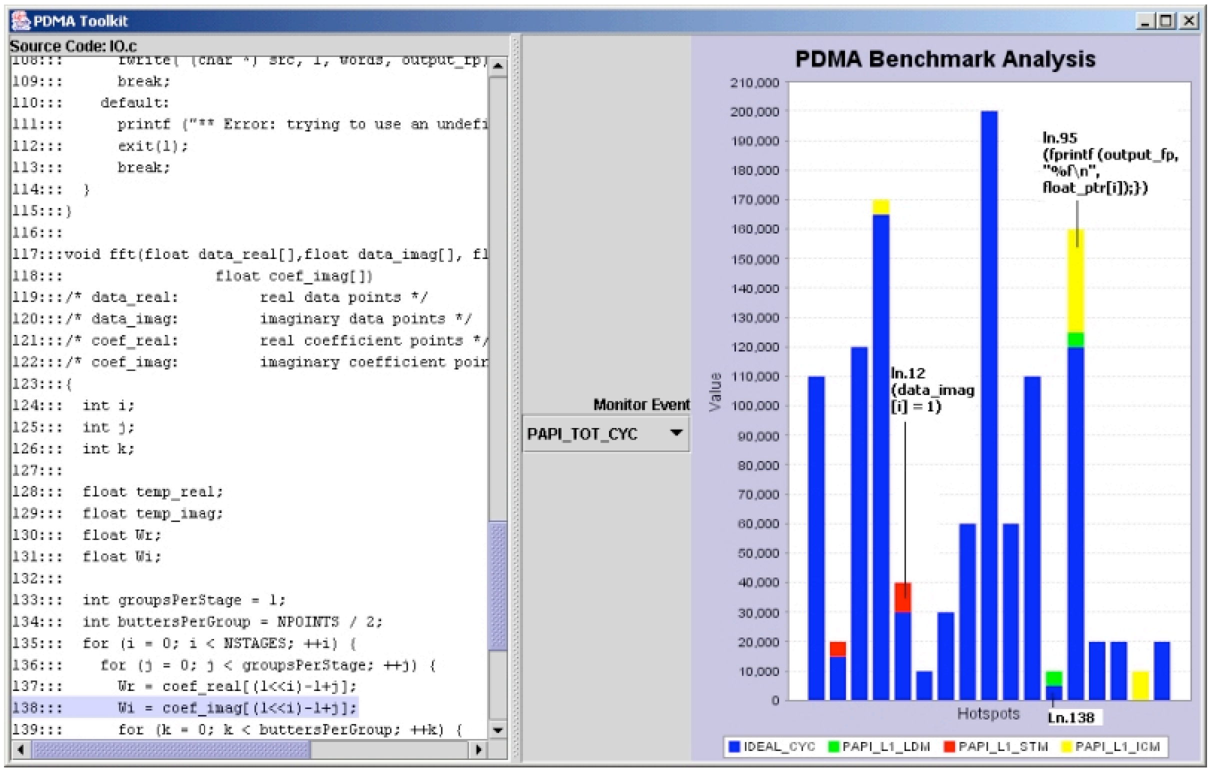


Source Code: compress.c

```
112:::   The result is returned in the same matrix as
113:::
114:::void dct(int block[B][B])
115:::{
116:::  int i;
117:::  int j;
118:::  int k;
119:::  float sum;
120:::
121:::  /* Multiply the input image block with the cos
122:::     in the temporary matrix "temp2d". */
123:::
124:::  for (i = 0; i < B; i++) {
125:::    for (j = 0; j < B; j++) {
126:::      sum = 0.0;
127:::      for (k = 0; k < B; k++) {
128:::        sum += block[i][k] * cos2[k][j];
129:::      }
130:::      temp2d[i][j] = sum;
131:::    }
132:::  }
133:::
134:::
135:::  /* Multiply the cosine matrix by the temporary
136:::     result back in the original matrix.  */
137:::
138:::  for (i = 0; i < B; i++) {  /* advances cos1 ro
139:::    for (j = 0; j < B; j++) {  /* no change */
140:::      sum = 0.0;
141:::      for (k = 0; k < B; k++) {  /* advances cos
142:::        sum += cos1[i][k] * temp2d[k][j] ;
143:::      }
```

Monitor Event: PAPI_L1_ICM

Figure 6. L1 Instruction Cache Misses for compress Benchmark



Source Code: IO.c

```
108:::      fwrite( (char *) src, 1, words, output_fp)
109:::      break;
110:::    default:
111:::      printf ("** Error: trying to use an undefi
112:::      exit(1);
113:::      break;
114:::  }
115:::}
116:::
117:::void fft(float data_real[],float data_imag[], fl
118:::                float coef_imag[])
119:::/* data_real:          real data points */
120:::/* data_imag:          imaginary data points */
121:::/* coef_real:          real coefficient points */
122:::/* coef_imag:          imaginary coefficient poir
123:::{
124:::  int i;
125:::  int j;
126:::  int k;
127:::
128:::  float temp_real;
129:::  float temp_imag;
130:::  float Wr;
131:::  float Wi;
132:::
133:::  int groupsPerStage = 1;
134:::  int buttersPerGroup = NPOINTS / 2;
135:::  for (i = 0; i < NSTAGES; ++i) {
136:::    for (j = 0; j < groupsPerStage; ++j) {
137:::      Wr = coef_real[(1<<i)-1+j];
138:::      Wi = coef_imag[(1<<i)-1+j];
139:::      for (k = 0; k < buttersPerGroup; ++k) {
```

Monitor Event: PAPI_TOT_CYC

Figure 7. Total Cycles' Stacked-Bar Chart for FFT Benchmark