Memory Profiling using Hardware Counters

Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki and Nicolai Kosche

{marty.itzkowitz,brian.wylie,christopher.aoki,nicolai.kosche}@sun.com

Sun Microsystems, Inc.
Menlo Park, California

ABSTRACT

Although memory performance is often a limiting factor in application performance, most tools only show performance data relating to the instructions in the program, not to its data. In this paper, we describe a technique for directly measuring the memory profile of an application. We describe the tools and their user model, and then discuss a particular code, the `MCF` *benchmark from SPEC CPU 2000. We show performance data for the data structures and elements, and discuss the use of the data to improve program performance. Finally, we discuss extensions to the work to provide feedback to the compiler for prefetching and to generate additional reports from the data.*

1. Introduction

Modern computer systems are using increasing numbers of ever faster processors to solve larger and larger problems. However, performance of those processors is limited by the need to supply data to them at ever increasing rates. A hierarchy of caches between the processors and main memory is used to improve performance: the processors run at full speed when using data from the caches closest to the processors, but are frequently stalled loading data from or storing data to the primary caches through secondary or tertiary caches and, ultimately, memory. Understanding how an application's data is structured in memory, and how it passes from memory through the cache hierarchy is one of the most important issues for understanding and improving the performance of applications on these systems.

In this paper, we describe extensions to the Sun ONE Studio™[1] compilers and performance tools [1] that can provide information relating to the data space of an application. The extension provides per-instruction details of memory accesses in the annotated disassembly, and provides data aggregated and sorted by object structure types and elements, a new observability perspective for application developers. The underlying framework can be a foundation for providing cache miss data to compilers and dynamic code generators, thus allowing cache-related optimizations.

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

1. Sun™, Sun ONE Studio™, Sun Fire™ 280R, and UltraSPARC® are trademarks of Sun Microsystems, Incorporated. Other names are trademarks of their respective owners.

In the rest of this introduction, we describe related work to measure application performance. In the next section, we describe the Sun ONE Studio compilers and tools, and their user model, and the UltraSPARC®-III hardware counters and their use in profiling. In the following section, we describe the `MCF` benchmark, part of the SPEC CPU 2000 suite of benchmarks, and the application of memory profiling to understanding the performance of that benchmark. Finally, we describe the opportunities for future work, and present our conclusions.

1.1 Related Work

Clock-based profiling has been available on UNIX systems for at least 20 years [2], and was implemented even earlier on CDC machines [3]. Hardware-counter based profiling was described by Zagha, *et.al*. at SuperComputing '96 in Pittsburgh [4].

Application profiling with processor counter overflows is supported by various commercial tools, such as Intel's VTune [5] and DEC/Compaq/HP's DCPI [6] and Caliper [7], and SGI's SpeedShop [8], as well as numerous academic tools, the latter often based on portable counter libraries such as PCL [9] and PAPI [10].

Each proprietary processor implements counters tailored for its particular architecture, with considerable variation in the number of counters supported, the events counted, whether an interrupt is provided on counter overflow and whether such an interrupt is precise or not, or a detailed history is available for sampled instructions. Alpha 21264 [6] and Pentium 4 [11] are examples of processors which support precise instruction and data address profiling.

The difficulty of collecting accurate profiles and correlating these with application program data structures has led to the use of simulators to acquire this information. While simulators can provide extremely detailed memory-reference information for both code- and data-oriented analyses, such as those offered by MemSpy [12], their runtime costs make them impractical for many large-scale applications. Buck and Hollingsworth [13] discuss cache-miss sampling, based on simulations of hardware that can report effective addresses. Data-object profiling/analyses may also be achieved through instrumentation of structure accesses (for example, by using Dyninst[14]), but these are also generally costly and intrusive. With address reference traces from suitably instrumented applications, the SIGMA infrastructure [15] can provide memory profiles and analyses for data-objects profiling.

Other work has been done to optimize data-layout via simulation and static analysis. The work of Chilimbi, *et.al*. [16] [17] [18] shows gains for commercial workloads. Chilimbi also has explored a mechanism to dynamically profile general-purpose programs and optimize them via prefetch insertion [19].

2. The Sun ONE Studio Compilers and Performance Tools

The Sun ONE Studio performance tools collect data based on clock- or hardware-counter overflow profiling, and present that information against functions, callers and callees, source lines and disassembly instructions. Attribution is against the elements in the text space of the application, but not against the data space representation. That allows the user to understand which parts of the program generate cache misses, but does not allow the user to understand which data objects are causing the misses.

This paper describes new functionality using hardware performance counters to examine memory system behavior in the context of the target's data space. Data-oriented analysis complements and extends existing performance analyses, providing observability into the performance of highly-optimized yet under-performing applications. Since this memory profile data is acquired from direct execution of uninstrumented applications, and since collection perturbation can be controlled through configuration of the processors' counter overflow rates, the tools are efficient and convenient to use even on large and complex applications.

The user model for the Sun ONE Studio performance tools consists of three steps: compiling the target program, collecting the data, and analyzing the data. These steps are discussed in the remainder of this section.

2.1 Compiling the Target Program

Normally, the user compiles the target program just as he or she would for production use, using the Sun ONE Studio compilers. For data space profiling, two flags, `-g -xhwcprof`, are needed. (An additional flag, `-xdebugformat=dwarf`, is used because DWARF symbol tables, but not the default STABS symbol tables, support memory profiling.) These two flags will cause changes to the symbolic information recorded with the executable: first, by ensuring that symbolic information about data references are written into the symbol tables; second, by cross-referencing each memory operation with the name of the variable or structure member being referenced; third, by adding information about all instructions that are branch targets; and fourth, by associating each instruction (PC) with a source line number.

Neither of these flags causes the compiler to suppress optimizations. For example, if both compiler options `-xprefetch` and `-xhwcprof` are specified, `-xhwcprof` does not suppress the optimizations enabled by `-xprefetch`. However, with `-xhwcprof` specified, the compiler does change the generated code slightly. It may add `nop` instructions between loads and any join-nodes (labels or branches) to help ensure that a profile event is captured in the same basic block as the triggering instruction. In addition, the compiler avoids scheduling load or store instructions in branch delay slots. With this compiler-generated padding, for some well-understood events, accuracies of nearly 100% have been observed. The impact of these modifications on performance is very much application-dependant, but generally minor. The runtime for the `MCF` application discussed below, as compiled with `-xhwcprof`, is approximately 1.3% greater than the runtime of the application compiled with identical flags, but without `-xhwcprof`.

2.2 Collecting the Data

Data collection is usually performed with the `collect` command, specifying data collection parameters along with the target program and any arguments the target program requires. Clock- and HW-counter profiling may be specified, independently of or along with tracing of synchronization delays, memory (heap) allocation and deallocation, or MPI messages. The result of a `collect` run is an experiment, which is a file-system directory with a `log` file giving a timestamped trace of high-level events during the run, a `loadobjects` file describing the target executable and any shared objects it uses, and additional files, one for each type of data recorded, containing the profile events and the callstacks associated with them.

To profile with hardware counters, the `collect` command takes a `-h` flag, which may specify either one or two counters; if two counters are requested, they must be on different registers. Counters may be specified by name, and an overflow interval may be specified as "`on`", "`high`", or "`low`" or as a numerical value. The settings are chosen to give overflow intervals corresponding to approximately 10 ms., 1 ms., and 100 ms. for the counter named "`cycles`"; for other counters, the time corresponding to the overflow value will obviously depend on the program behavior. The intervals are chosen as prime numbers, to reduce the probability of correlations in the profiles.

2.2.1 The UltraSPARC-III Hardware Counters

On UltraSPARC-III family of chips (see, for example, [20]), there are two counter registers, each of which can count one of a number of events. The counter may be preloaded with a value, and when the counter overflows, a interrupt is generated. That interrupt is translated by the Solaris™ Operating Environment into a signal (`SIGEMT`) delivered to the process being profiled. The signal is received by a handler that is part of the data collection system, and causes a data record to be written by that handler for the event.

Counters are available for Cycles, Instructions Completed, Instruction-cache (I$) Misses, Data-cache (D$) Read Misses, Data-translation-lookaside-buffer (DTLB) Misses, External-cache (E$) References, E$ Read Misses, E$ Stall Cycles, and many others. Some of the counters count events (that is the number of times the particular trigger occurred), but others count cycles; for cache counters, the counters that measure in cycles are especially interesting, since they count the actual time lost because of the events, not just the number of events. The `collect` command, if run with no arguments, will generate a list of available counters for the machine on which it is run.

2.2.2 Counter Skid

As is the case with many other chips, the UltraSPARC-III does not always deliver a precise trap when a HW counter overflows. Since a counter may overflow quite late in the pipeline, the logic to deliver a precise trap is complex, and may significantly affect the performance of the chip in normal operation. Instead, the trap may be delivered after the event that caused the counter overflow and may arrive quite a bit later.

At the time the signal arrives, the PC that is delivered with it represents the next instruction to issue, which is rarely the instruction following the one that caused the counter overflow. A performance measurement tool should take this skid into account in recording and interpreting the data. The hardware does not capture the address of the data whose reference causes a memory-related counter overflow event—only the register set at the time of the signal delivery is reported.

2.2.3 Apropos Backtracking Search

As mentioned above, although a PC is reported when the profile interrupt is delivered, the instruction at the location pointed to by the PC is generally not the one triggering the event—that instruction may not yet have been executed. The instruction immediately preceding it in execution may be the one causing the event, but counter skid adds a great deal of uncertainty.

To address this problem we implemented an apropos backtracking search. It is specified by the user by prepending a "+" to the name of a memory-related counter used for profiling. During data collection, the collector walks back in the address space from the PC reported with the overflow until it finds a memory-reference instruction of the appropriate type. The actual PC of the instruction causing the event is called the trigger PC; the PC determined by apropos backtracking is referred to as the candidate trigger PC.

The first memory reference instruction preceding the PC in address order may not be the first preceding instruction in execution order. In particular, if there is any instruction that is a branch target, there is no way to determine which path to the PC was taken, so the true trigger PC can not be determined. It is too expensive to locate branch targets at data collection time, so the candidate trigger PC is always recorded, but it is validated during data reduction.

Once the collector has backtracked to find the candidate trigger PC, it can disassemble the instruction, and determine which registers are used to compute the effective address. However, the contents of the registers may have been changed by intervening instructions while the counter is skidding, so that even if the expression to compute the effective address is known, it may not be computable. The collector makes that determination, and either reports a putative effective address, or indicates that the address could not be determined.

2.3 Analyzing the data

As mentioned above, the result of a data-collection run is an experiment. It may be analyzed by a GUI program (`analyzer`) or by a command-line equivalent (`er_print`). Both use the same shared-object to process the data.

For all experiments, the data is reduced to an annotated representation of the program graph, with performance metrics for each node in the graph. The nodes correspond to PCs, and the graph corresponds to the dynamic call graph of the program. The reduced data can be used to show a function list, to show callers and callees of a function, with information about how the performance metrics are attributed to the callers and callees, and to show annotated source or disassembly code of the target.

For HW-counter experiments with apropos backtracking, additional data structures are built corresponding to the data objects referenced by the target. The node in the program graph corresponds to an instruction, and, for memory reference instructions, the symbol tables are used to determine the name of the data object being referenced.

In order to validate a candidate trigger PC, the data reduction process must first verify that there were no branch targets between the next PC as delivered with the counter overflow signal, and the candidate trigger PC determined at data collection time. If there was an intervening branch target,

the analysis code can not determine how the code got to the point of the interrupt, and so can not be sure which instruction caused the event. The data analysis inserts an artificial branch-target PC, and attributes the events to that artificial branch target.

3. Example: The MCF SPEC CPU 2000 Benchmark

The MCF application, part of the SPEC CPU 2000 benchmark suite, was developed by Löbel [21] for his dissertation. It is a single-depot vehicle scheduler formulated as a large-scale minimum-cost flow problem solved with a network simplex algorithm accelerated with column generation. It is neither multithreaded nor parallelized, but it does stress the performance of a single processor.

3.1 Experimental setup

The MCF application consists of 11 source files in the C language. The program was compiled using the Sun ONE Studio 8 C Compiler (v5.5) with the `-fast -xarch=v9` build options along with the options for memory profiling support: `-g -xhwcprof -xdebugformat=dwarf`.

Two experiments were collected using the Sun ONE Studio 8 Performance Tools `collect` command on the MCF application. They were run on a dual 900 MHz UltraSPARC-III Cu Sun Fire 280R™ system with 2GB of RAM; each processor has 64kB of 4-way-associative level-1 D$, organized in 32-byte lines, and 8MB of 2-way-associative level-2 E$, organized in 512-byte lines. The machine was running Solaris 9 update 4.

The two experiments were collected with the following command lines:

```
collect -S off -p on -h +ecstall,lo,+ecrm,on mcf.exe mcf.in
collect -S off -p off -h +ecref,on,+dtlbm,on mcf.exe mcf.in
```

The first experiment collects both clock profile data (`-p on`) and hardware counter profile data for E$ Read Misses and E$ Stall Cycles; the second experiment collects hardware counter profile data for E$ References and DTLB Misses. Apropos backtracking search for memory-reference instructions and their effective data addresses was specified for all four HW counters, using a "+" preceding the counter name for each counter.

3.2 Discussion

The remainder of this section will discuss the performance data recorded from these two experiments.

3.2.1 Performance Metrics

Figure 1 shows the performance metrics recorded from the two experiments for the artificial function `<Total>`, which represents the sum across all functions in the program. The program as a whole is almost 100% CPU-bound, but spends more than half its time stalled for E$ Misses. Furthermore, estimating the cost of a DTLB Miss as 100 cycles suggests an additional cost of 28 seconds, or another 5% of the total run time. The overall E$ Read Miss rate for the application is about 6.4%. Clearly, performance of the memory subsystem is a dominant factor in the overall performance of the application.

3.2.2 Functions

Figure 2 shows the function list, with the exclusive time and percentage for User CPU time and E$ Stall Cycles, and the percentages of E$ Read Misses, E$ References and DTLB Misses. It shows that over 95% of the application's User CPU Time is spent in the top three functions: `refresh_potential` (51%), `primal_bea_mpp` (23%) and, `price_out_impl` (22%).

The topmost function, `refresh_potential`, is responsible for about half the total User CPU time, and disproportionately more E$ Stall Cycles (62%) and DTLB Misses (88%). It is

```
            Exclusive Total LWP Time:      552.677  secs.
             Exclusive User CPU Time:      549.404  secs.
           Exclusive System CPU Time:        3.082  secs.
             Exclusive Wait CPU Time:        0.180  secs.
            Exclusive User Lock Time:        0.     secs.
      Exclusive Text Page Fault Time:        0.     secs.
      Exclusive Data Page Fault Time:        0.     secs.
           Exclusive Other Wait Time:        0.010  secs.
           Exclusive E$ Stall Cycles:      297.569  secs.
                             " count: 267812254774
            Exclusive E$ Read Misses:   1580927631
                   Exclusive E$ Refs:  24873218652
              Exclusive DTLB Misses:    256265124
```

Figure 1. Performance metrics for the `<Total>` function

Excl. User CPU		Excl. E$ Stall Cycles		Excl. E$ Read Misses %	Excl. E$ Refs %	Excl. DTLB Misses %	Name
sec.	%	sec.	%				
549.404	100.0	297.569	100.0	100.0	100.0	100.0	<Total>
280.706	51.1	184.224	61.9	62.3	38.4	88.0	refresh_potential
127.319	23.2	90.267	30.3	29.6	13.8	9.4	primal_bea_mpp
120.134	21.9	11.389	3.8	4.0	41.7	0.8	price_out_impl
6.154	1.1	5.533	1.9	2.0	0.3	0.2	flow_cost
4.933	0.9	0.011	0.0	0.	3.4	0.	sort_basket
3.733	0.7	2.711	0.9	1.0	0.4	0.1	dual_feasible
2.682	0.5	1.656	0.6	0.6	0.8	0.1	update_tree
1.861	0.3	0.933	0.3	0.2	0.4	0.3	primal_iminus
0.690	0.1	0.411	0.1	0.1	0.0	0.7	write_circulations
. . .							

Figure 2. The Function List

responsible for 62% of all the E$ Read Misses, but only 38% of the E$ references; it has an E$ Read Miss rate of 10.3%. Conversely, the third function, `primal_bea_mpp`, is responsible for 42% of all E$ references, but only 4% of E$ Read Misses; it has a cache-miss rate of 0.6%. Clearly, memory usage is a source of performance problems for this application in general, and for the topmost functions in particular.

To understand the critical code in these `MCF` functions, source and disassembly listings annotated with metrics associated with each source line or instruction can be examined.

3.2.3 Annotated Source and Disassembly

Figure 3 shows an excerpt from the annotated source of the most expensive function, showing User CPU Time and E$ Stall Cycles for the critical loop. The critical loop contains numerous pointer dereferences which make it difficult to determine the most costly ones. However, the annotated disassembly will allow us to understand those references.

Figure 4 shows an excerpt from the annotated disassembly of the same function. The critical loop in `refresh_potential` consists of fewer than 30 instructions (including `nop`s added for padding). Approximately half of the instructions in the critical loop are loads and stores. Four of these memory-referencing instructions (all loads) are particularly costly: three of them each account for approximately 10% of the total E$ Stall Cycles (or 5% of the total run time), while the first in the loop at 0x1000031B0 accounts for almost 20% (10% of total runtime). Almost 90% of all the DTLB misses for the entire program also occur in this loop, with about one-fourth of them being caused by loads of `node.orientation` and the remainder coming from `arc.cost`.

Closer examination of the annotated disassembly reveals several features specific to data-oriented analysis.

First, the E$ Stall Cycles metric correlates quite well with memory-referencing instructions; the

```
  Excl.     Excl. E$
 User CPU     Stall
    sec.   Cycles sec.
. . .
   0.        0.          79.     tmp = node = root->child;
   0.        0.          80.     while( node != root )
                         81.     {
   0.290     0.          82.        while( node )
                         83.         {
## 37.826   61.978       84.          if( node->orientation == UP )
## 75.993   49.745       85.            node->potential = node->basic_arc->cost
                                                + node->pred->potential;
                         86.          else /* == DOWN */
                         87.          {
## 60.552   41.311       88.            node->potential = node->pred->potential
                                                - node->basic_arc->cost;
   3.202     0.          89.            checksum++;
                         90.          }
                         91.
   0.560     0.          92.          tmp = node;
## 75.343   29.500       93.          node = node->child;
                         94.        }
                         95.
   0.220     0.          96.        node = tmp;
. . .
```

Figure 3. Annotated source of critical loop of `refresh_potential`

metric usually appears on a load instruction, suggesting that the apropos backtracking correctly determined the trigger PC. On the other hand, significant amounts of User CPU time is shown against unlikely instructions, such as the 57 secs. associated with the `add` instruction at 0x1000031D8 and 48 secs. associated with the `sub` instruction at 0x100003208. Clock profiling events, which are the basis of the User CPU Time metric, are delivered with PC of the instruction next to be executed and can not be corrected with apropos backtracking, but memory reference events can be corrected. To some extent, the correlation is the fortuitous behavior of this particular example code. If there were floating-point instructions immediately following the load instruction, the skid would be greater.

Second, lines marked with an asterisk and labeled `<branch target>`, are included in the listing and some have an E$ Stall Cycles metric associated with them. For example, the line labeled 0x100003218* preceding the instruction at 0x100003218 shows 1.2 secs. of E$ Stall Cycles which clearly are not attributable to the branch instruction itself. These metrics represent events where the apropos backtracking was blocked because of the control transfer target. In this example, the metric values are not statistically significant and can be ignored; there is no way to tell whether the overflow events came via the branch at 0x1000031E8 or simply fell through from the immediately preceding block of instructions.

Third, data-object descriptor names are shown as annotations for memory-referencing instructions, based on the information provided by the compiler.

From these data it is readily apparent that two of the costly reads are for `arc.cost` (with no other references to `structure:arc` in the critical loop) while the third is for `node.orientation`, (with none of the references to other elements of `structure:node` being particularly costly). The initial reference to `node.orientation` at the start of the loop brought the structure into the cache where other elements of it could subsequently be efficiently accessed. The references to `arc.cost` occur immediately after their address was determined (since they are pointer elements of the previously loaded `structure:node`): too soon to be effectively prefetched.

3.2.4 PCs

Figure 5 shows a list of those PCs with the highest performance metric values in the application,

```
  Excl.   Excl. E$   Excl.
User CPU  Stall      DTLB
  sec.  Cycles sec. Misses %
 . . .
   0.      0.         0.     [82] 1000031a8* <branch target>        <===----<<<
   0.290   0.         0.     [82] 1000031a8: be,pn  %xcc,0x100003220
   0.      0.         0.     [82] 1000031ac: nop
   0.      0.522      0.     [84] 1000031b0* <branch target>        <===----<<<
#  5.514  61.456     24.3    [84] 1000031b0: ldx    [%o3 + 56], %o2
                                 {structure:node -}.{long orientation}
# 75.303  28.300      0.1    [93] 1000031b4: ldx    [%o3 + 24], %o4
                                 {structure:node -}.{pointer+structure:node child}
  32.313   0.         0.     [84] 1000031b8: cmp    %o2, 1
   0.      0.         0.     [69] 1000031bc: nop
   0.      0.         0.     [84] 1000031c0: bne,pn %xcc,0x1000031f0
   0.560   0.         0.     [92] 1000031c4: mov    %o3, %o5
   0.040   3.056      0.     [85] 1000031c8: ldx    [%o3 + 16], %o0
                                 {structure:node -}.{pointer+structure:node pred}
   3.583   2.767      0.0    [85] 1000031cc: ldx    [%o3 + 64], %g4
                              {structure:node -}.{pointer+structure:arc basic_arc}
   4.953   0.578      0.2    [85] 1000031d0: ldx    [%o0 + 88], %g5
                                 {structure:node -}.{cost_t=long potential}
# 10.687  43.345     29.9    [85] 1000031d4: ldx    [%g4 + 32], %g1
                                 {structure:arc -}.{cost_t=long cost}
# 56.720   0.         0.     [85] 1000031d8: add    %g1, %g5, %g2
   0.010   0.         0.2    [85] 1000031dc: stx    %g2, [%o3 + 88]
                                 {structure:node -}.{cost_t=long potential}
   0.      0.         0.     [69] 1000031e0: nop
   0.881   0.         0.     [69] 1000031e4: nop
   0.      0.         0.     [85] 1000031e8: ba     0x100003218
   0.      0.         0.     [93] 1000031ec: cmp    %o4, 0
   0.      0.         0.     [88] 1000031f0* <branch target>        <===----<<<
   0.      2.689      0.     [88] 1000031f0: ldx    [%o3 + 16], %g4
                                 {structure:node -}.{pointer+structure:node pred}
   3.202   0.         0.     [89] 1000031f4: inc    %g3
   0.      0.         0.     [93] 1000031f8: cmp    %o4, 0
   0.      2.367      0.0    [88] 1000031fc: ldx    [%o3 + 64], %g5
                              {structure:node -}.{pointer+structure:arc basic_arc}
   3.362   0.956      0.     [88] 100003200: ldx    [%g4 + 88], %g2
                                 {structure:node -}.{cost_t=long potential}
#  9.537  35.300     32.8    [88] 100003204: ldx    [%g5 + 32], %g1
                                 {structure:arc -}.{cost_t=long cost}
  47.643   0.         0.     [88] 100003208: sub    %g2, %g1, %o2
   0.010   0.         0.1    [88] 10000320c: stx    %o2, [%o3 + 88]
                                 {structure:node -}.{cost_t=long potential}
   0.      0.         0.     [69] 100003210: nop
   0.751   0.         0.     [69] 100003214: nop
   0.      1.200      0.     [93] 100003218* <branch target>        <===----<<<
   0.040   0.         0.     [93] 100003218: bne,pt %xcc,0x1000031b0
   0.530   0.         0.     [69] 10000321c: mov    %o4, %o3
 . . .
```

Figure 4. Annotated disassembly of critical loop of `refresh_potential`

ranked by E$ Read Misses.

Examination of the PCs ranked by their E$ Read Misses shows that although the single top PC comes from `primal_bea_mpp`; the next four PCs all come from `refresh_potential`. (The poorer correlation of User CPU profile data with critical instructions is also evident.)

3.2.5 Data Objects

Figure 6 shows a list of the data object types in the program, annotated with the cost of the data references to each.

```
Excl. User      Excl. E$        Excl.      Excl.       Name
CPU             Stall Cycles    E$ Read    DTLB
   sec.     %     sec.     %   Misses %   Misses %
549.404 100.0  297.569 100.0   100.0      100.0    <Total>
  1.841   0.3   63.567  21.4    24.8        4.7    primal_bea_mpp + 0x000002EC
                                 {structure:arc -}.{long ident}
  5.514   1.0   61.456  20.7    21.4       24.3    refresh_potential + 0x000000D0
                                 {structure:node -}.{long orientation}
 10.687   1.9   43.345  14.6    16.3       29.9    refresh_potential + 0x000000F4
                                 {structure:arc -}.{cost_t=long cost}
  9.537   1.7   35.300  11.9    13.4       32.8    refresh_potential + 0x00000124
                                 {structure:arc -}.{cost_t=long cost}
 75.303  13.7   28.300   9.5     7.9        0.1    refresh_potential + 0x000000D4
                                 {structure:node -}.{pointer+structure:node child}
  0.280   0.1    9.644   3.2     2.8        1.9    primal_bea_mpp + 0x0000030C
                                 {structure:node -}.{cost_t=long potential}
 14.270   2.6    5.000   1.7     1.8        1.5    primal_bea_mpp + 0x00000310
                                 {structure:node -}.{cost_t=long potential}
  3.583   0.7    2.767   0.9     1.0        0.0    refresh_potential + 0x000000EC
                                 {structure:node -}.{pointer+structure:arc basic_arc}
  0.120   0.0    1.944   0.7     1.0        0.5    price_out_impl + 0x000004C8
                                 {structure:arc -}.{flow_t=long flow}
  0.010   0.0    2.422   0.8     1.0        0.1    flow_cost + 0x00000120
                                 {structure:arc -}.{long ident}
  0.030   0.0    2.333   0.8     1.0        0.0    dual_feasible + 0x0000001C
                                 {structure:arc -}.{flow_t=long flow}
  3.683   0.7    3.111   1.0     0.9        0.     price_out_impl + 0x000004CC
                                 {structure:arc -}.{flow_t=long flow}
 24.797   4.5    2.367   0.8     0.9        0.0    price_out_impl + 0x000002CC
                                 {structure:node -}.{long time}
  0.       0.    2.367   0.8     0.9        0.0    refresh_potential + 0x0000011C
                                 {structure:node -}.{pointer+structure:arc basic_arc}
  0.370   0.1    1.567   0.5     0.7        0.2    price_out_impl + 0x000002C4
                                 {structure:arc -}.{pointer+structure:node tail}
  0.       0.    2.689   0.9     0.7        0.     refresh_potential + 0x00000110
                                 {structure:node -}.{pointer+structure:node pred}
  0.040   0.0    3.056   1.0     0.6        0.     refresh_potential + 0x000000E8
                                 {structure:node -}.{pointer+structure:node pred}
  . . .
```

Figure 5. PCs, ranked by E$ Read Misses

```
Data.  E$       Data.      Data.     Data.
Stall Cyc.     E$ Read     E$        DTLB
   sec.   %    Misses %    Refs %    Misses %   Name

297.569 100.0   100.0       100.0     100.0     <Total>
166.402  55.9    59.4        37.3      70.0     {structure:arc -}
124.601  41.9    39.5        41.4      29.7     {structure:node -}
  6.378   2.1     1.2        19.0       0.2     <Unknown>
  3.467   1.2     1.1         2.1       0.1     (Unspecified)
  2.611   0.9     0.0         5.3       0.      (Unresolvable)
  0.200   0.1     0.1         0.5       0.1     (Unascertainable)
  0.178   0.1     0.0         2.2       0.      {structure:basket -}
  0.100   0.0     0.0        11.2       0.      (Unidentified)
  0.011   0.0     0.          0.        0.      {structure:network -}
  0.      0.      0.0         0.0       0.      <Scalars>
  . . . .
```

Figure 6. Data objects, ranked by E$ Stall Cycles

The data objects associated with memory-referencing instruction events also provide a basis for a fundamentally different set of metrics derived from the hardware counter profile event data. (Such data-derived metrics are not possible from other hardware counter events or other types of profile

data.) Aggregation by data object reveals that 63% of E$ Read Misses and Data TLB Misses relate to accesses to `structure:arc` versus 36% from accesses to `structure:node`. Negligible numbers of misses are attributable to other structures.

The `<Unknown>` data object shows metrics corresponding to about 2% percent of the E$ Stall Cycles, and 19% of all E$ References. E$ References have significantly greater skid than the other memory metrics in these experiments, and they are also far more widely distributed, thus accounting for the difference. Many such references come from libraries such as `libc.so.1`, that were not compiled with the appropriate compiler flags for memory profiling.

The `<Unknown>` object is an aggregate of several types of indeterminate references: `(Unspecified)`, meaning that the compiler did not give a symbolic reference for the instruction; `(Unresolvable)`, meaning that the apropos backtracking could not determine the trigger PC; `(Unascertainable)`, meaning that some modules that had memory events were not compiled with the `-xhwcprof` flag; and (`Unidentified)`, meaning that the compiler did not identify the data object, which is most likely a compiler-temporary. One additional subset of `<Unknown>` does not happen to appear in the data from these experiments: `(Unverifiable)`, meaning that the compiler branch-target information provided was inadequate to determine the validity of the trigger PC.

The effectiveness of the apropos backtracking for each metric is approximately equal to 100% minus the metric values associated with the `(Unresolvable)` and `(Unascertainable)` data objects. From Figure 6, the scheme can be seen to be more than 99% effective for E$ Stall Cycles, almost 100% effective for E$ Read Misses, 100% effective for DTLB misses (which are precise), and almost 94% effective for E$ References (which have the greatest skid associated with them). We have found approximately the same effectiveness for these in experiments on a large commercial application.

Figure 7 shows the expansion of the data object `structure:node`, showing metrics for each

Data. E$ Stall Cycles sec.	%	Data. E$ Read Misses %	Data. E$ Refs %	Data. DTLB Misses %	Name +offset .field-name
124.601	41.9	39.4	41.5	29.8	{structure:node -}
0.056	0.0	0.0	0.	0.0	+0 .{long number}
0.	0.	0.	0.	0.	+8 .{pointer+char ident}
7.056	2.4	1.3	1.8	0.5	+16 .{pointer+structure:node pred}
29.100	9.8	8.2	3.7	0.1	+24 .{pointer+structure:node child}
0.444	0.1	0.0	2.2	0.0	+32 .{pointer+structure:node sibling}
0.	0.	0.	0.0	0.	+40 .{pointer+structure:node sibling_prev}
0.244	0.1	0.1	0.4	0.2	+48 .{long depth}
61.523	20.7	21.4	8.0	24.3	+56 .{long orientation}
5.256	1.8	2.0	1.5	0.1	+64 .{pointer+structure:arc basic_arc}
0.078	0.0	0.0	0.1	0.	+72 .{pointer+structure:arc firstout}
0.033	0.0	0.	0.0	0.	+80 .{pointer+structure:arc firstin}
18.133	6.1	5.2	16.7	4.5	+88 .{cost_t=long potential}
0.256	0.1	0.1	0.2	0.	+96 .{flow_t=long flow}
0.056	0.0	0.0	1.4	0.	+104 .{long mark}
2.367	0.8	0.9	5.5	0.0	+112 .{long time}

Figure 7. Dataobject `structure:node` expansion

element in the structure.

From this table, it can be seen that of the 42% of E$ Stall Cycles related to accessing the `node` structure, with the bulk of the cost attributable to accessing its `child`, `orientation` and `potential` elements, at offsets of 24, 56 and 88 bytes respectively, in the 120-byte structure. Further investigation reveals that the majority of the references to elements `node.orientation` and `node.child` occur in function `refresh_potential`, while the majority of the references to element `node.potential` occur in function `primal_bea_mpp`. The 32-byte separation between these elements places them in different D$ lines (each 32 bytes wide), which is likely to limit cache effectiveness.

If the first of the dynamically-allocated 120-byte `structure:node` data objects is E$-line aligned, the fifth such data object will be split over two 512-byte E$ lines. In fact, 28% of these 120-byte data objects end up split this way. This suggests that it would be more efficient to re-arrange the elements of `structure:node` to pack the most referenced elements contiguously into 32 bytes to facilitate re-use in the D$, and either pad the structure with an additional 8 bytes or otherwise align them such that only whole data objects are mapped into E$ lines.

Similarly, of the 56% of E$ Stall Cycles attributable to accessing `structure:arc`, 27% are due to element `cost` (mostly in `refresh_potential`) and 22% are due to element `ident` (mostly in `primal_bea_mpp`). While these constitute the critical references, accounting for the bulk of the E$ Read Misses and E$ Stall Time, other references which show up in the profile should be considered in any possible optimization. The current performance data is insufficient to quantify the number of such references which may be efficiently exploiting the E$ but may be adversely impacted by code or data modifications.

The application also shows high DTLB Misses, which contribute about 5% to the total run time. The analysis suggests that rebuilding with pages larger than the system default page size of 8kB will improve performance by decreasing the number of DTLB Misses, thus making more effective use of the DTLB.

3.3 Performance Improvements based on the Analysis

The first change suggested by the data above is padding the `node` structure with an additional 8 bytes, aligning `node` and `arc` structures on cache lines, and re-arranging the members of the `node` and `arc` structures according to their frequency of reference. Doing so and recompiling produced a 16.2% speedup in `MCF` total execution time.

The second change suggested is to change the page size used for the heap segment. Doing so, by adding a `-xpagesize_heap=512k` flag and recompiling, produced a 3.9% gain.

Combining the changes produced a cumulative decrease of 20.7% in the `MCF` application total execution time.

4. Future Work

The current implementation presents information about data structures and members, and the profiles attributed to them. Future work is under consideration to support feedback to the compiler to allow it to produce more-efficient executables, and to support reports and displays of additional information.

Since the experiments contain the information necessary to know which memory references cause the cache-misses, the data can be used to construct a feedback file, allowing a recompilation of the target to be done with the insertion of prefetch instructions.

Event data addresses can be further analyzed by corresponding machine entities, such as the memory segment (of load objects or allocated to stack, heap, shared or otherwise mapped memory), and broken down by page for those segments. Alternatively, addresses can be aggregated by corresponding cache line for analysis.

In the current implementation, although the actual effective address of the data causing a cache miss is determined (where possible), it is only used in showing an individual cache-miss event in the analyzer. Future work could support translating the effective addresses into structure object instances, and aggregating data by instance, rather than only by type.

5. Conclusions

We have discussed the issues concerning memory accesses, and their effects on performance, summarizing related work on memory profiling. We then discussed the Sun ONE Studio compilers

and performance tools, and described the new functionality added to support memory profiling. We used the tools to explore the behavior of the MCF benchmark, and showed how they can highlight the performance issues relating to memory accesses. We then showed how the analysis suggested changes to the code that improve its performance by more than 20%. Finally, we discussed extensions to the present work to further explore memory behavior in applications.

REFERENCES

[1] *Program Performance Analysis Tools*, Sun Microsystems, Inc. Publication 817-0922-10, May 2003.

[2] S.L.Graham, P.B.Kessler, and M.K.McKusick, "An Execution Profiler for Modular Programs," *Software Practice and Experience*, **13**, 671-685, August 1983.

[3] D.F.Stevens, SPY for the CDC 6600, private communication, 1968.

[4] Marco Zagha, Brond Larson, Steve Turner and Marty Itzkowitz, "Performance Analysis using the MIPS R10000 Performance Counters," *Proceedings of SuperComputing '96*, Pittsburgh, PA, November, 1996.

http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/ZAGHA.PS

[5] K. Sridharan, "VTune: Intel's Visual Tuning Environment," *Proceedings of USENIX-NT '97*, 11 August, 1997.

[6] Jennifer Anderson, Lance Berc, George Chrysos, Jeffrey Dean, Sanjay Ghemawat, Jamey Hicks, Shun-Tak Leung, Mitch Lichtenberg, Mark Vandevoorde, Carl A. Waldspurger, and William E. Weihl, "Transparent, Low-Overhead Profiling on Modern Processors," *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT'98, Paris, France), 13 October, 1998.

http://research.compaq.com/SRC/dcpi/papers/pfdc98.ps

[7] Robert Hundt, "HP Caliper: A Framework for Performance Analysis Tools," *IEEE Concurrency*, **8**, 64-71, Oct-Dec 2000

[8] *SpeedShop User's Guide (IRIX 6.5)*, Silicon Graphics, Inc., manual 007-3311-005, 1998

[9] Rudolf Berrendorf and Bernd Mohr, *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*, Technical Report IB-9816, Forschungszentrum Jülich.

http://www.fz-juelich.de/zam/PCL

[10] Shirley Browne, Jack Dongarra, Nathan Garner, Kevin London, and Philip Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," *Proceedings of SC2000*, Dallas, Texas, November 2000.

http://www.sc2000.org/techpapr/papers/pap.pap256.pdf

[11] Brinkley Sprunt, "The Performance Monitoring Features of the Pentium4 Processor," submitted to *IEEE Micro*, February 2002.

http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/doc/pentium4_emon.pdf

[12] Margaret Martonosi, Anoop Gupta and Thomas Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.1-12, May 1992.

http://portal.acm.org/citation.cfm?doid=133057.133079

[13] Bryan R. Buck and Jeffrey K. Hollingsworth, "Using Hardware Performance Monitors to Isolate Memory Bottlenecks." *Proceedings of SC2000*, Dallas, TX, November 2000.

http://www.supercomp.org/sc2000/Proceedings/techpapr/papers/pap.pap197.pdf

[14] *DyninstAPI Programmer's Guide*, University of Maryland College Park, release 4.0, May 2003.

http://dyninst.org/docs/dyninstProgGuide.v40.pdf

[15] Luiz De Rose, K. Ekanadham, Jeffrey K. Hollingsworth and Simone Sbaraglia, "SIGMA: A Simulator

Infrastructure to Guide Memory Analysis," *Proceedings of SC2002*, Baltimore, MD, November 2002.

http://www.sc-2002.org/paperpdfs/pap.pap191.pdf

[16] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi, "An Efficient Profile-Analysis Framework for Data-Layout Optimizations," *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002, pp 140-153

[17] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, "Cache-Conscious Structure Layout," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999, 1-12

[18] Trishul M. Chilimbi, Bob Davidson, and James R. Larus, "Cache-Conscious Structure Definition," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999, 13-24

[19] Trishul M. Chilimbi and Martin Hirzel, "Dynamic Hot Data Stream Prefetching for General-Purpose Programs," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002, 199-209

[20] Sun Microsystems, Inc., *UltraSPARC III Cu User's Manual*, April 2003.

http://www.sun.com/processors/manuals/USIIIv2.pdf

[21] Andreas M. Löbel, "Optimal Vehicle Scheduling in Public Transit," Ph.D. thesis, Technische Universität Berlin, 1997. SPEC CPU2000 181.mcf; source code shown with permission of the author.