

Performance Analysis Using the MIPS R10000 Performance Counters

Marco Zaghera
Brond Larson
Steve Turner
Marty Itzkowitz

Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94043-1389
{marcoz,brond,swt,martyi}@sgi.com

Abstract:

Tuning supercomputer application performance often requires analyzing the interaction of the application and the underlying architecture. In this paper, we describe support in the MIPS R10000 for non-intrusively monitoring a variety of processor events -- support that is particularly useful for characterizing the dynamic behavior of multi-level memory hierarchies, hardware-based cache coherence, and speculative execution. We first explain how performance data is collected using an integrated set of hardware mechanisms, operating system abstractions, and performance tools. We then describe several examples drawn from scientific applications, which illustrate how the counters and profiling tools provide information that helps developers analyze and tune applications.

Keywords:

performance analysis, profiling tools, hardware performance counters, MIPS R10000, SGI Power Challenge

1. Introduction

A fundamental question asked by HPC application developers is: "Where is the time spent?". A likely follow-up question -- a much more challenging question -- is "How is the time spent?". This question is only partially addressed by software-based analysis tools. Approaches such as simulation [1,13] can generate accurate counts of processor and memory-system events, but cause a slowdown of several orders of magnitude for accurate multiprocessor simulations [25,28]. Approximate analysis such as static predictions generated from instrumented programs [5, 9,12], while less expensive, perturbs the behavior of the executable, and may fail to capture the dynamic behavior of the computation. With the increasing use of dynamic, superscalar instruction scheduling (e.g., on the MIPS R10000 and HP PA-8000), static estimates of computation are inherently less accurate, and with the increasing importance of memory-system behavior, the lack of memory-system statistics is particularly limiting -- especially for supercomputing applications.

The R10000 design addresses these problems by providing hardware support for counting various types of events, such as cache misses, memory coherence operations, branch mispredictions, and several categories of issued and graduated instructions. These counters are useful to application developers for gaining insight into application performance and for pinpointing performance bottlenecks. In addition to application tuning, counters have many other uses, including predicting application performance and scalability for future processors and memory systems, analyzing architectural tradeoffs, generating address traces or address statistics, analyzing bus or communication traffic [15,17,23], evaluating compiler transformations, profiling the kernel [24], testing hardware, and characterizing workloads [7,8,27].

Similar types of hardware counters have appeared in other processors. Counters have been used extensively on Cray vector processors [2], and appear, in some form, in microprocessors such as the Intel Pentium [16], IBM Power2 [14,26], DEC Alpha [3], and HP PA-8000 [11]. However, most of the microprocessor vendors provide documentation on counters and counter-based performance tools primarily to hardware developers and selected performance analysts. We have a different philosophy about the purpose of counters -- one that is most closely aligned with the approach traditionally taken at Cray Research. We believe that counter data is important for application developers, and that counters should be documented and supported by high-level tools. Our design constraints differed from Cray in several key areas, however. In addition to having tighter (area) budget constraints, we felt it was necessary to support procedure-level event profiling for arbitrary binaries, without requiring explicit recompiling or relinking, and without perturbing memory addresses.

We were able to satisfy these constraints using integrated design of hardware, operation system functionality, and performance tools. These components are described in this paper, and are illustrated using several representative applications.

The remainder of the paper is organized as follows. In Section 2 the R10000 architecture is summarized, focusing on the parts of relevance for performance counters. This section also covers software support at the operating system level. Section 3 describes the user-mode software tools by which the counter facility is made available. Several pedagogical examples drawn from scientific and engineering applications are discussed in Section 4. Conclusions are summarized together with a brief discussion of future directions in Section 5. Although the system will be presented from the bottom up, many of the lower level details are relegated to appendices. Readers interested only in application level tuning can skim sections 1 and 2.

2. Hardware and Operating System support

2.1 Hardware Support

The MIPS R10000 [18,19] is a 4-way superscalar microprocessor that currently operates at a clock frequency of 195 MHz. It is able to fetch four instructions from its 32 kilobyte on-chip instruction cache on each processor cycle. It uses speculative execution to continue decoding instructions beyond conditional branches (up to 4 unresolved branches). By issuing instructions to its 2 integer and 2 floating-point functional units out of order, and using register renaming to reduce data dependencies, it is capable of continuing to make forward progress in program execution while some instructions wait for data dependencies or cache misses to be satisfied. The 32 kilobyte, 2-way set-associative on-chip data cache is itself capable of issuing 4 requests to the external data cache (1 MB and 2MB in initial

R10000-based systems) and/or the system interface while continuing to service other accesses.

All of these capabilities combine to allow the R10000 to be very flexible in dealing with problems that would normally stall the execution pipeline of a more conventional microprocessor. However, this flexibility comes at a price in terms of user-visible state. Code scheduling and data placement decisions are much more difficult to make when the processor itself seeks to adapt to less-than-ideal circumstances.

The performance counter facility helps compensate for these additional complications by providing system designers, compiler writers and application developers with detailed information on the behavior of the chip. In order to make the counters feasible to implement with negligible impact on overall processor performance, a small set of events was chosen. These events were selected such that they form a simple, orthogonal set of events from which software analysis could produce a more sophisticated and easily-interpreted description of system behavior. The events were chosen with particular attention to those that would illuminate the dynamic behavior of the chip, since static analysis can be more effectively implemented in other ways. We wanted to bring out aspects of program behavior that would otherwise be difficult, if not impossible, to see.

The state of the performance counters is visible to applications via a register, which can be read and written via kernel-level processes. The hardware implements two 32-bit counters, each of which can be configured to monitor any one of 16 distinct event types [20]. A total of 30 different event types can be monitored, because two event types ("Cycles" and "Graduated instructions") may be monitored by either of the two counters. The full set of counters is described in Appendix A.

In addition to accumulating values that can be read from the performance counter registers, the counters provide a kernel-level interrupt that is triggered when one of the counters overflows. This facility has proved to be very valuable, because it allows the counters to generate interrupts that are periodic in any specified event and it allows the period to be controlled in software by setting the initial counter value after each overflow.

The performance counter facility was designed to be non-intrusive. Because the monitoring is done in hardware rather than in software, it is possible to extract detailed information about the state of the processor without affecting the behavior of the program being monitored. The facility was designed such that it introduced no critical timing paths into the chip circuitry, so that the performance of the hardware was not degraded by their inclusion.

The logical circuitry required to implement the counter facility was specified in under 300 lines of RTL, and implemented with fewer than 5,000 transistors -- less than 0.2% of the total number of non-cache transistors included in the R10000. The logic was created primarily via synthesis rather than in full-custom circuitry because the timing issues were not critical. The only difficult aspect of the circuit implementation was routing the wires required to carry the signals that are monitored back to the central counting circuitry. Because the signals of interest are scattered across the logical blocks that make up the R10000, several long wires were required. Figure 1 illustrates the wiring required to implement the counters.

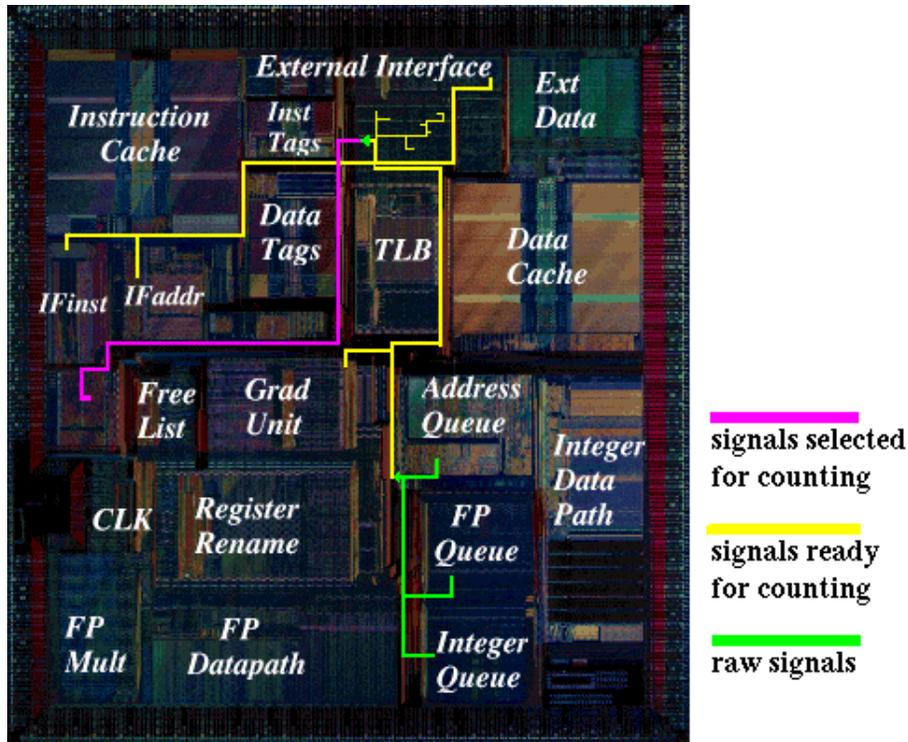


Figure 1: Wiring scheme for performance counters.

The most expensive aspect of implementing the performance counters was not the die area, transistor count or wires routed. The expense was dominated by the design time required first to select an appropriate set of performance events, then to determine and/or derive a corresponding set of on-chip signals, and finally to verify the correctness of the counters. This architectural specification, logic design and verification process resulted in a design with very low cost in terms of the per-chip resources used. Such a design is eminently suitable for inclusion in a product that is to be replicated hundreds of thousands of times.

2.2 Operating system and library support

The operating system (OS) virtualizes the shared counter hardware and presents an abstraction that is a more flexible and sharable (in user mode). The OS layer sits above the hardware and has exclusive direct access to the counter hardware, since it is accessed only through supervisor-only coprocessor instructions. Since the OS has sole access it must provide an interface that is simple and efficient, introducing only the minimum necessary overhead. The focus of the interface is correspondingly modest in terms of ease of use and presentation of results, preferring to leave these to higher layers of software in exchange for efficiency. The abstraction includes two modes for counting, user mode and system mode. We focus on user mode in this paper. Details of the OS interface and system profiling are described in Appendix B.

The kernel maintains up to 32 different 64-bit virtual counters for the user program, which are presented to the developer through a programming interface. The number 32 is chosen to allow all events to be specified simultaneously, even though the underlying hardware counters can only count pairs of events at one time, constrained by which event types are available on a particular counter (see Appendix A).

Thus a user program can specify up to 16 events per hardware counter. The kernel switches or *multiplexes* events across scheduler interrupt (clock tick) boundaries as needed to count the specified events co-resident on each physical counter. The interpretation of the multiplexed counts is statistical since each of any n co-resident events per physical counter will be counted during $1/n$ of the time quanta. This introduces noise in exchange for coverage. It has been successful for screening because, for event categories responsible for significant fractions of program time, the relative uncertainties in the sampled counts are empirically small.

The OS interface also allows a developer to specify a frequency for each event, upon which the kernel synchronously delivers a user signal to the process. This capability has enabled various experiments such a memory address tracing, through the design of appropriate handlers for cache miss overflow interrupts.

An easy to use interface built on top of the OS interface is provided in the *perfex* library. The library manages common-case bookkeeping, allowing occasional users to access the counters without learning somewhat unfamiliar `ioctl`-based system calls.

3. Developer Tools

We currently provide two tools that present R10000 counter data: *perfex*, a command-line interface, and SpeedShop, the new generation of SGI performance tools. *perfex* provides program-level event information using a simple command line interface (much like the Cray `hpm` utility [2]). The user may either monitor two compatible event counters, or use the OS multiplexing features to sample all the counters. (An example of *perfex* use is listed in Appendix C.) For multithreaded applications *perfex* reports data for each thread in addition to aggregate counts. *perfex* can identify which processor events are causing performance problems, and SpeedShop performance tools then identify the parts of the program source encountering them.

The SpeedShop performance tools provide for a wide variety of traditional types of performance data, all with a common usage model and data format. SpeedShop was extended for the R10000 to provide statistical profiling based on overflows from the hardware counters. The data recorded is a histogram of "hits" vs. program counter (PC), where each hit corresponds to one overflow of a particular counter. The data recorded is analogous to that recorded in a traditional PC sampling run. It can show where in the source of the program the overflows are being triggered, at the function and approximate source-line level.

PC sampling, based on clock interrupts, runs some risk of undesirable correlation with the behavior of the application program. The operating system itself uses the same clock as the profiler to schedule tasks, including the application; some applications are awakened at scheduler ticks, and for those, PC sampling will systematically over- or under-count those routines that get triggered by scheduling (depending on whether the OS schedules first and then profiles, or vice-versa). Hardware counter profiling can bypass this correlation. By using cycle counter overflows, we obtain essentially the same data as clock tick profiling, but since the OS does not use the cycle counter overflow for anything else, it won't exhibit the same correlation.

We can also use overflows of a counter other than cycles to obtain information previously completely unavailable. For example profiling can be based on I-cache misses, producing a histogram indicating which parts of the program are thrashing the I-cache. Profiling based on D-cache misses can show which

instructions are triggering the misses, which may suggest restructuring array accesses, etc. These experiments are also at some risk of correlation: if every 1000th D-cache miss were profiled in a program with a loop that incurs 100 misses in each iteration, all the counts will go to the 100th miss site, rather than being uniformly distributed across all 100 sites. We provide two sets of preset overflow values for each of the counters; each value is a prime number, and thus profiles that are similar for both overflow values should be statistically valid. The meticulous user can run additional experiments with different overflow numbers to ensure that the data are statistically valid. (Examples of SpeedShop use are presented in Appendix C.)

Some additional extensions exploiting the counter overflow functionality are under investigation. The current profiling is done entirely within the OS, which makes the profiling very light-weight. As mentioned in Section 2, the OS allows an overflow-based profiling signal to be delivered to the application. With this functionality the user callstack could be unwound on interrupt, allowing a profile of inclusive cache misses. The experiment in data space profiling based on this functionality has already been alluded to in Section 2. Combining this memory-address indexed miss information with the program symbol management in SpeedShop could show which elements of each array are causing the cache misses.

4. Examples

In this section, we step through the process of tuning application programs using examples from operations research, computation fluid dynamics, image processing, and weather modeling. In order to simplify the presentation of some examples, we analyze smaller kernels that have similar behavior to the original applications. In each case we highlight relevant hardware features, software tools, and program restructuring techniques. All experiments were performed on an SGI Power Challenge 10000 [21]. Graphs were produced from performance data generated with `perfex`, the `perfex` library, and SpeedShop.

4.1 Cache-miss Profiling

Our first example demonstrates a typical, straightforward use of the counter-based performance tools. The application optimizes personnel requirements through improved resource scheduling. We can estimate the influence of various hardware resources by running `perfex` in multiplexing mode to sample all the counters and multiplying the projected event counts by typical event costs. As indicated in Figure 2, the time is dominated by secondary data cache misses:

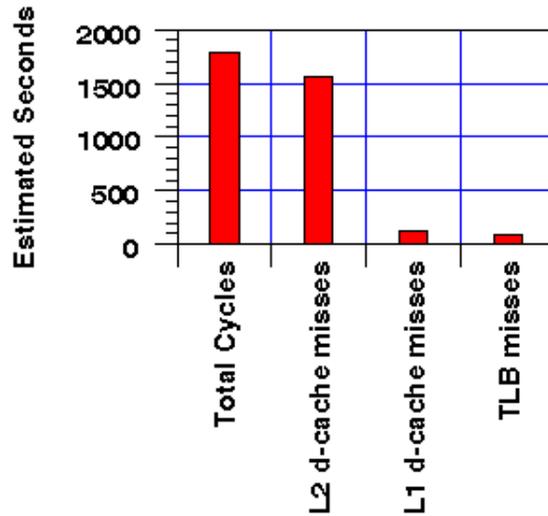


Figure 2: Estimates of event costs in the original program. Secondary data cache misses account for the majority of cycles.

We can refine the profiling by using SpeedShop to generate line-level estimates of secondary data cache misses. Figure 3 shows that a few concentrated areas of the source code account for the majority of secondary cache misses, and in fact, a single source line accounts for over 40% of the misses. The poor performance on this source line is caused by a search loop that accesses a four-byte integer field in each element of an array of structures, where each structure is larger than a single 128-byte secondary cache line. By storing the integer items in a separate shadow array, the secondary cache miss rate of the entire application is reduced from 30% to 12%.

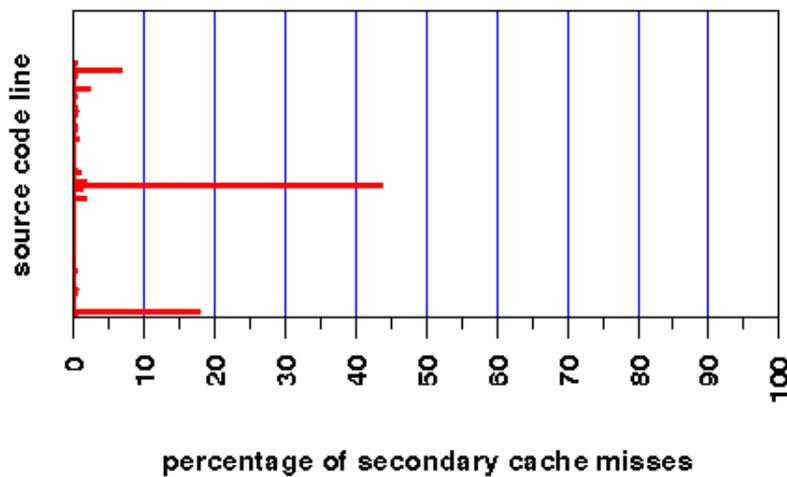


Figure 3: Approximate percentage of the total number of secondary data cache misses that occur at each source code line. Each bar represents a single line in the 45,000-line source program.

4.2 Multi-level Memory Hierarchy Profiling

We next consider a simple example that mimics the memory behavior of an Alternating Direction Implicit (ADI) method for solving a set of three-dimensional partial differential equations. This class of applications is typically very memory bandwidth intensive, and our sample kernel highlights memory performance by using a simplified computational kernel while retaining representative memory access patterns. In each iteration, the algorithm performs one "sweep" along each dimension of an $N \times N \times N$ grid. There are data dependences between neighboring elements along the sweep dimension, but no dependences across the dimensions orthogonal to the sweep direction.

With some initial timings on a $100 \times 100 \times 100$ grid, we observe that the running time is not as good as anticipated (based on floating-point operation counts), and in addition, we observe that the processing rate varies considerably with small changes in the grid size. Using SpeedShop to profile the cycle count, we discover that the X, Y, and Z sweeps consume 26%, 28%, and 45% of the cycles respectively. By running perfex in multiplexing mode, we find that the time is dominated by memory operations: approximately 45% of the time for translation lookaside buffer (TLB) misses, 45% for second level data cache misses, and 5% for first level data cache misses.

With additional SpeedShop experiments for these 3 memory events, we get much closer to pinpointing the problems. Figure 4 shows the performance for the X, Y, and Z sweeps as a function of grid size.

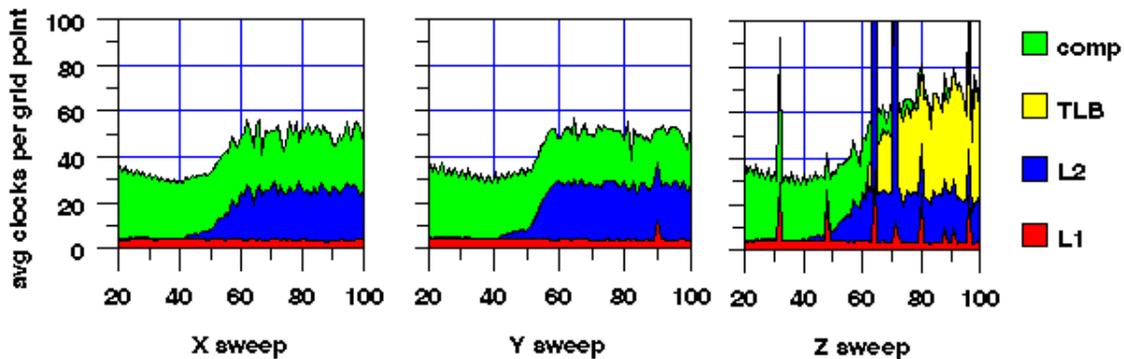


Figure 4: Time for X, Y, and Z sweeps of ADI as a function of grid size.

The total time (represented by the top curve) is broken down into components for primary and secondary cache misses, TLB misses, and other computation (including integer and branch operations, as well as floating operations). Component times were estimated by multiplying event counts by a typical event cost, assuming no overlap between events.

Several performance problems are apparent from the graphs. They can be remedied with several optimizations:

array padding

The spikes are caused by cache thrashing. Assuming Fortran array allocation conventions, an array element, $\text{grid}(i, j, k)$, with declared dimensions N_x by N_y by N_z maps to a linear array index of $i + N_x * j + N_x * N_y * k$. In the worst case, when $N_x * N_y$ is a large power of two, each data element along the Z axis maps to the same cache line. The cache thrashing can be eliminated by padding the

leading array dimensions.

loop interchange

Notice that TLB misses dominate the time for the Z sweep. The original implementation performs the Z sweep with independent passes over each of the N_x*N_y lines along the Z axis. Each pass accesses memory with a large stride (approximately N_x*N_y grid elements), a stride that exceeds the 16-Kbyte page size. Thus, thrashing results when the capacity of the 64-entry TLB is exceeded. The TLB performance can be improved by interchanging the inner loop over the Z axis with one (or both) of the outer loops over orthogonal dimensions. That is, passes along lines of the Z axis are interleaved such several contiguous grid points with the same Z coordinate are accessed before accessing the next page. This causes a slight increase in the number of primary cache misses due to the increased working set, but greatly reduces the number of TLB misses.

loop fusion

In the original program, the X and Y sweeps are organized such that the loop over the Z dimension is outermost. A straightforward optimization is to fuse the outer loops of the X and Y sweeps, i.e., to process one Z-slice of the grid at a time, first performing all the X sweeps within the slice, then all the Y sweeps. This greatly reduces the cost of the Y sweeps, since most of the data needed is loaded into the cache during the X sweep.

The result of these three optimizations is shown in Figure 5.

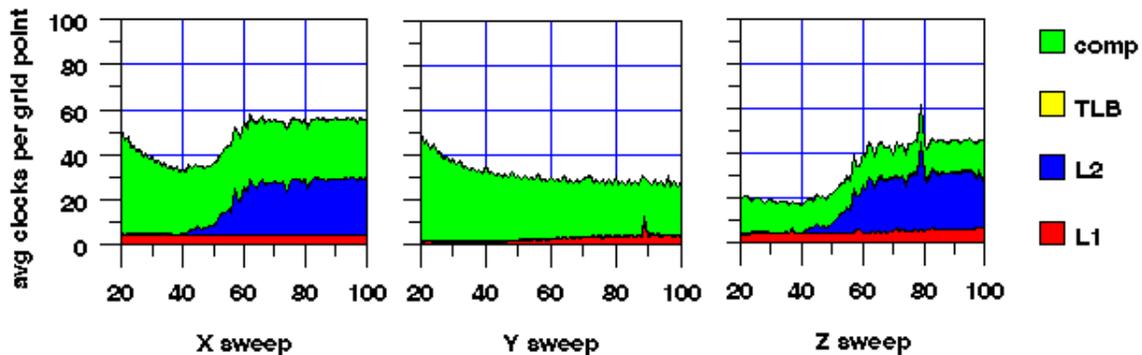


Figure 5: Time for optimized X, Y, and Z sweeps as a function of grid size.

Overall, these improvements lead to performance that is more consistent and is approximately a factor of 1.3 faster than the original implementation on large problem sizes.

4.3 Multiprocessor Sharing

The Challenge and Power Challenge systems maintain cache coherency using a cache-snooping protocol [6]. Memory coherence transactions can be monitored using two types of events: interventions and invalidations. These transactions provide a mechanism for other components in a multiprocessor system to modify the state of data cached in an R10000 microprocessor. Both types of request are communicated to the processor through the system bus.

In this example, we analyze the multiprocessor performance of a weather modeling program. The program uses spectral methods and the kernel we analyze is from the physics loop. The program has

acceptable performance on one processor, but scales poorly to more processors. Using `perfex` in multiplexing mode, it becomes clear that the running time is dominated by secondary cache misses. In order to understand this behavior, we examine counters related to the secondary cache.

We first observe that the number of secondary data cache misses *increases* as the number of processors is increased. Furthermore, the total number of interventions and invalidations closely tracks the number of secondary cache misses, indicating a high degree of sharing. For the parallel algorithm used in this application, very little sharing is expected. Each processor works on an independent region of memory and accumulates a sum of its results. Using SpeedShop, we find that most of the cache misses occur in the accumulation step, indicating that false sharing [4] is probably the cause of the performance problem. Upon further analysis of the source code, we see that the sums for each processor are allocated contiguously, and thus, multiple sums are located on the same cache line. This problem can be fixed with a small change in the array declaration or with appropriate compiler directives. After eliminating the false sharing, we see a dramatic performance improvement in the number of coherency events (as shown in Figure 6) and in the parallel speedup (as shown in Figure 7).

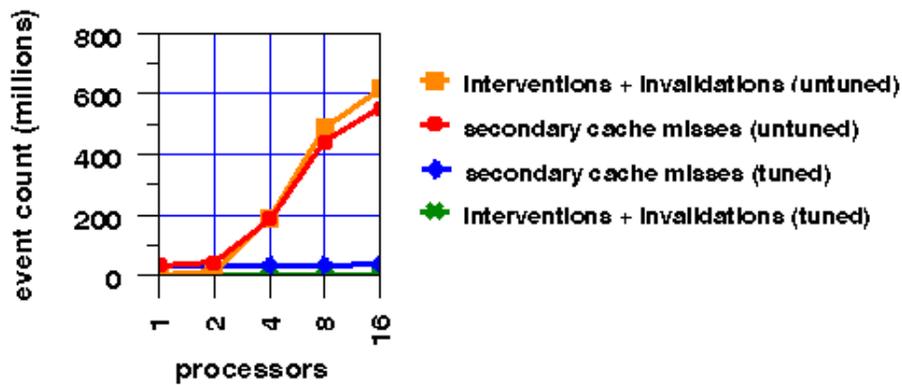


Figure 6: Cache event counts vs. number of processors for an untuned version with false sharing, and for a tuned version.

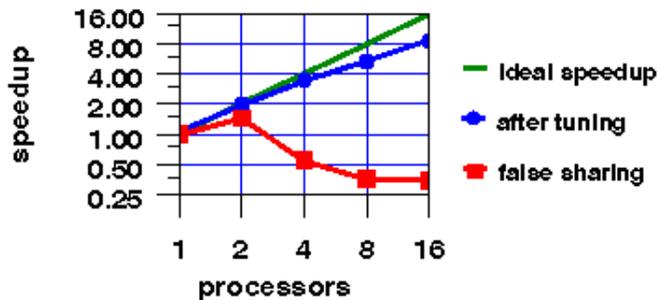


Figure 7: Speedup for an untuned version with false sharing, and for a tuned version.

4.4 Branch Prediction

In this example, we analyze the performance of a general-purpose, two-dimensional convolution subroutine. This example demonstrates how the performance counter tools can be used to quickly identify performance problems, and to gain insight into the performance interaction of the application, compiler, and hardware.

In this particular application, we use a 3 by 3 filter for blurring an image. By running `perfex`, we see that our initial version exhibits several performance problems: many cycles per floating point operation, many loads per floating point operation, and interestingly, many mispredicted branches. See Figure 8.

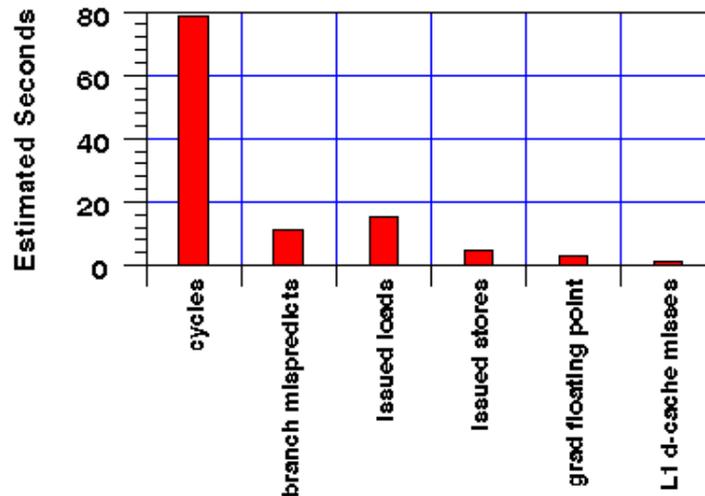


Figure 8: Estimates of event costs in the original convolution subroutine.

By examining the counters for mispredicted branches and issued branches, we compute that 34% of the branches are mispredicted. This statistic leads us to a quick diagnosis. The mispredictions are caused by the short innermost loops over the convolution mask (3x3). The compiler is unable to remove the branches without restructuring the code, since the subroutine was written for an arbitrary mask size and was compiled separately.

Understanding the importance of branch prediction requires some more background on the processor architecture. In the R10000, the path taken by a conditional branch is predicted using a two-bit branch history memory, which is updated after a branch decision is resolved. For nested DO loops, the hardware will typically predict that all branches on the innermost loops are taken, and thus will repeatedly make an incorrect prediction on the last iteration of the inner loop. With the short inner loops used in this program, the misprediction rate is expected to be near one-third.

In this case, there is a simple solution. If the convolution routine is recompiled with loop nest optimization (LNO) enabled, the compiler interchanges the loops such that the trip count on the inner loop will be derived from the image dimensions rather than the mask dimensions, and "unrolls-and-jams" the outer loop to enable common subexpression elimination of memory references. In addition, enabling inter-procedural analysis (IPA) allows the compiler to propagate the constant bounds on the mask size and reduce the number of instructions issued.

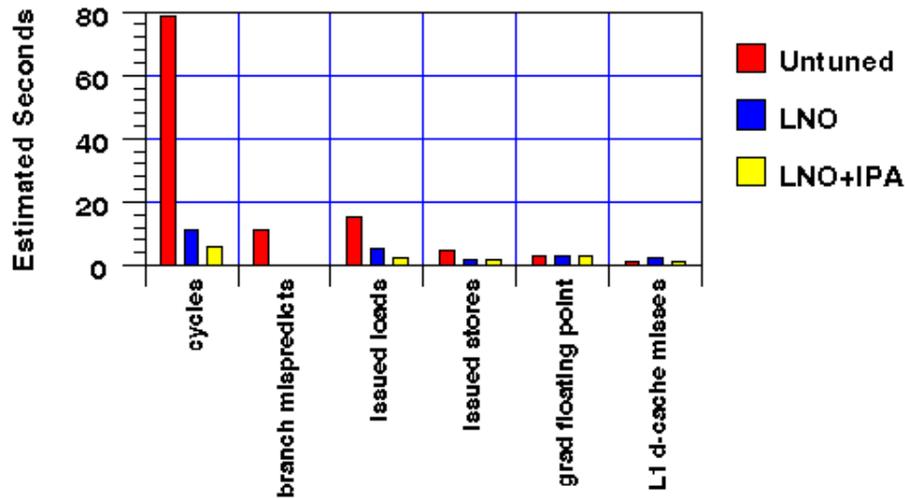


Figure 9: Performance improvements from loop nest optimization and inter-procedural analysis.

As indicated in Figure 9, not only have the branch mispredictions been virtually eliminated, but in addition, the number of issued instructions has been greatly reduced. Overall, the set of improvements introduced by changing compiler flags improves the cycle count by a factor of 13.5.

5. Discussion

Two main conclusions emerge from our experience with the design, implementation, and use of hardware performance counters and counter-based profiling tools.

First, **the counters were well worth the silicon and design time to include them.** They have provided valuable insight into dynamic program behavior for developers of applications, compilers, operating systems and hardware. Counters provide a wealth of data about resource use during program execution and this data is just beginning to be organized and used effectively. Benefits deriving from users enhanced ability to quantify and report performance bugs may be one of the hidden bonuses.

Second, the simplest way for any counter facility to justify its cost is in terms of customer application developer use. Given this, **performance tools are critical**, since the raw events constitute at best an "assembly language" of performance experiments. These performance primitives need to be combined, or "compiled", into performance experiments that answer application-level questions and allow the resulting data to be presented visually and correlated with application program components and name spaces.

Future development of the hardware counters are likely to respond to the experiences of application developers and to trends in microprocessor design. In future designs, we would like to include additional accumulators and event types, where the chip design and chip area costs can be justified. New events of particular interest are memory stall cycles, outstanding miss counts, and prefetch hits and misses. Along with this capability, some mechanism for measuring correlations of events---such as conditionalizing the count of one event on the occurrence of another---would be helpful.

The utility of these enhancements lies in being able to address questions such as the following:

- "How much did cache misses stall the processor?"
- "How effective was prefetching?"
- "How much do branch mispredictions cost?"
- "What was the distribution of memory latencies?"

Making the tool software more flexible and extensible should be another goal. A SpeedShop API would allow special purpose modules to use SpeedShop's management of source symbols, data file format, and eventually its graphical interface.

We are looking forward to receiving more customer feedback from both novice and expert users. This feedback will guide development of our hardware and software tools. Viewed as the first of many possible self-informing [10] architectural features, counters represent an exciting capability, with a wealth of unexplored potential.

Appendix A: R10000 performance counters

The following table lists the events that can be measured on the two event counters. *Note:* These event definitions correspond to all versions of the processor older than revision 3.0. For details on the exact meaning of the events and the definitions for later revisions, see the R10000 Microprocessor User's Manual [20].

Event	Counter 0	Counter 1
0	Cycles	Cycles
1	Instructions issued	Instructions graduated
2	Load/prefetch/sync/CacheOp issued	Load/prefetch/sync/CacheOp graduated
3	Stores issued	Stores graduated
4	Store conditionals issued	Store conditionals graduated
5	Failed store conditionals	Floating-point instructions graduated
6	Branches decoded	Quadwords written back from primary data cache
7	Quadwords written back from secondary cache	TLB refill exceptions
8	Correctable ECC errors on secondary cache data	Branches mispredicted
9	Instruction cache misses	Data cache misses
10	Secondary cache misses (instruction)	Secondary cache misses (data)
11	Secondary cache bank mispredictions (instructions)	Secondary cache bank mispredictions (data)
12	External intervention requests	External intervention request hits
13	External invalidate requests	External invalidate request hits
14	Virtual coherency conditions	Stores or prefetch-exclusives to <i>Clean Exclusive</i> secondary cache blocks
15	Instructions graduated	Stores or prefetch-exclusives to <i>Shared</i> secondary cache blocks

Appendix B: OS interface and system profiling

The OS provides an interface for acquiring and releasing the counters, reading the counts, or modifying the counting mode or event. This interface takes the process ID as a parameter and allows one process to manipulate the counters for another process, as long as the manipulating process is a member of the same process group (or is root). (The OS counter abstraction is presented to the user primarily through the `/proc` interface, and `ioctl` calls.)

For a process using the counters in user mode, the control block for the counters is kept in the user area. Thus, once a process forks, the child starts life with the same counter state as its parent. Counter values for the child are zeroed at the fork. When the child exits, if the kernel sees that the parent is waiting for the child it will add the child's 64-bit counts to those of the waiting parent. (It is also possible to cause these counts to be returned to a non-waiting parent using one of the ioctls to modify the parent's state.) Thus in the normal case a parent, on completing, will contain the aggregate counts of all of its descendants.

System mode profiling is quite different, and incompatible with user mode. System mode is useful for profiling the kernel or monitoring the behavior of a job mix. It can be used with SpeedShop or with system monitoring tools such as Performance Co-Pilot [22], which can be configured to show a graphical representation of various counters on a per-CPU basis.

System mode is defined by a state in which the process has root privileges and the kernel mode bit in the counter control register is set. In this mode there are no context-switch boundaries and therefore other programs will be unable to use the counters when they are being used in system mode. System mode has higher priority than user mode, and processes with root privileges can forcibly acquire the counters with a special system call. User programs that lose the counters in this way, even if the counters are subsequently restored to usability, are given a mechanism to detect this fact.

To support use of the counters in system mode, each CPU has its own control block for the counters, pointed to in its private area. There is also a global counter control block which maintains counter state for the entire system. When the counters are being used in system mode they are not read and stored across context switch boundaries. Unless they are explicitly read by a program, the counters are not read by the kernel until there is an overflow interrupt. When this occurs, the CPU on which the interrupt occurred updates its own private virtual counters, and no changes are made to the global counter control block.

When the counters are read in system mode via the `ioctl` call, the per-CPU counters are aggregated into the global counters such that they reflect the total of all the counted events for the entire system. This same aggregation of the per-CPU counters happens when the counters are released.

Appendix C: Sample output from `perfex` and `prof`

Here is an example of `perfex` output for a molecular dynamics simulation of Silicon. This example uses the multiplexing feature of the OS interface. (Notice for example that the two counts of graduated instructions disagree by approximately 1% due to multiplexing projection noise).

```
% perfex -a md < md_input_file
[...program output...]
Cycles..... 7945053664
Issued instructions..... 11008061056
Issued loads..... 2871087744
Issued stores..... 459886640
Issued store conditionals..... 0
Failed store conditionals..... 0
Decoded branches..... 888399408
Quadwords written back from scache..... 1077760
Correctable scache data array ECC errors..... 0
Primary instruction cache misses..... 231856
```

```

Secondary instruction cache misses..... 15264
Instruction misprediction from scache way prediction table... 67440
External interventions..... 198528
External invalidations..... 91472
Virtual coherency conditions..... 0
Graduated instructions..... 8975754960
Cycles..... 7945053664
Graduated instructions..... 8847468640
Graduated loads..... 2421679696
Graduated stores..... 433456128
Graduated store conditionals..... 0
Graduated floating point instructions..... 3132175344
Quadwords written back from primary data cache..... 26034192
TLB misses..... 11504
Mispredicted branches..... 128178704
Primary data cache misses..... 25746192
Secondary data cache misses..... 236000
Data misprediction from scache way prediction table..... 578176
External intervention hits in scache..... 198064
External invalidation hits in scache..... 90880
Store/prefetch exclusive to clean block in scache..... 181744
Store/prefetch exclusive to shared block in scache..... 19568

```

The next example shows SpeedShop output for the same program, from an experiment where graduated floating point (gfp) instructions were the sampling trigger.

To produce this output, the following commands were executed:

```
% ssrun -exp gfp_hwc md < md_input_file
```

produces the file md.gfp_hwc.12133. Then,

```
% prof md.gfp_hwc.12133<br>
```

produces

```
-----
Profile listing generated Wed May 1 16:25:50 1996
with:      prof md.gfp_hwc.12133
-----
```

```

Counter          : Graduated fp instructions
Counter overflow value: 32771
Total number of ovfls : 96520
CPU               : R10000
FPU               : R10010
Clock             : 195.0MHz
Number of CPUs    : 1

```

```
-----
-p[rocedures] using counter overflow.
Sorted in descending order by the number of overflows in each procedure.
Unexecuted procedures are excluded.
-----
```

```

overflows(%)  cum overflows(%)  procedure (dso:file)
50772( 52.6)  50772( 52.6) compute_forces_IK (md:...)
28081( 29.1)  78853( 81.7)      __pow (/usr/lib64/libm.so:...)
7323(  7.6)   86176( 89.3)      __exp (/usr/lib64/libm.so:...)
4262(  4.4)   90438( 93.7) min_image_distance_x (md:...)
1973(  2.0)   92411( 95.7) build_neighbor_lists (md:...)
1190(  1.2)   93601( 97.0) integrate_positions (md:...)
1094(  1.1)   94695( 98.1) integrate_velocities (md:...)
1012(  1.0)   95707( 99.2) min_image_distance_y (md:...)
637(  0.7)    96344( 99.8) min_image_distance_z (md:...)
169(  0.2)    96513(100.0) clean_up_positions (md:...)
6(  0.0)      96519(100.0)      __cos (/usr/lib64/libm.so:...)
1(  0.0)      96520(100.0)      _mixed_dtoa (/usr/lib64/libc.so.1:...)

```

96520(100.0)

TOTAL

Here is a profile of the same program based on equally-spaced secondary cache data misses as a trigger. The cache miss intensive procedures are not particularly well correlated with either floating point instruction count or execution time, a reflection of the fact that while this code has very few overall cache misses, the majority are in routines accounting for a minor fraction of the total execution time.

...

-p[rocedures] using counter overflow.
Sorted in descending order by the number of overflows in each procedure.
Unexecuted procedures are excluded.

overflows(%)	cum overflows(%)	procedure (dso:file)
1623(46.2)	1623(46.2)	integrate_velocities (md:...)
820(23.3)	2443(69.5)	compute_forces_IK (md:...)
509(14.5)	2952(84.0)	__pow (/usr/lib64/libm.so:...)
285(8.1)	3237(92.1)	integrate_positions (md:...)
69(2.0)	3306(94.1)	min_image_distance_x (md:...)
57(1.6)	3363(95.7)	build_neighbor_lists (md:...)
37(1.1)	3400(96.7)	__exp (/usr/lib64/libm.so:...)
22(0.6)	3422(97.4)	compute_v2sum (md:...)
13(0.4)	3435(97.7)	total_momentum_check (md:...)
11(0.3)	3446(98.0)	_times (/usr/lib64/libc.so.1:...)
10(0.3)	3456(98.3)	clean_up_positions (md:...)
10(0.3)	3466(98.6)	min_image_distance_y (md:...)
9(0.3)	3475(98.9)	dump_state (md:...)
5(0.1)	3480(99.0)	min_image_distance_z (md:...)

...

Acknowledgments

Many people contributed to the design and implementation of the counter hardware and support tools, notably, Robert Nuckolls, Ken Yeager, John Mashey, Jun Yu, Stephen Belair, and Jeff Fier. Kevin Kuhn extracted the signal trace layout include in Figure 1. Jeff Fier suggested the ADI and convolution examples, and Gilles Garcia provided the weather kernel. Ed Rothberg and Jonathan Hardwick provided valuable feedback on early drafts of this paper.

References

[1] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. **PROTEUS: A High-Performance Parallel-Architecture Simulator**. MIT/LCS/TR-516, MIT 1991.

<file://ftp.lcs.mit.edu/pub/supertech/papers/TR516-proteus.ps.Z>

[2] Cray Research Inc. **UNICOS Performance Utilities Reference Manual**. Cray Research Publication SR-2040. January, 1994.

[3] Digital Equipment Corporation. **pfm - The 21064 Performance Counter Pseudo-Device**. DEC OSF/1 Manual pages. 1995.

[4] S. J. Eggers and T. E. Jeremiassen. **Eliminating False Sharing**. International Conference on Parallel Processing, 377-381. August 1991.

<http://netlib.att.com/netlib/att/cs/home/jeremiassen/papers/icpp-91.html>

[5] A. Eustace and A. Srivastava. **ATOM: A Flexible Interface for Building High Performance Program Analysis Tools**. DEC WRL Technical Note TN-44. July 1994.

<http://www.research.digital.com/wrl/publications/abstracts/TN-44.html>

[6] M. Galles and E. Williams. **Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor**. Proceedings of the 27th Annual Hawaii International Conference on System Sciences, 1994. http://www.sgi.com/Technology/challenge_paper.html

[7] H. Gao and J. L. Larson. **Workload Characterization Using the Cray Hardware Performance Monitor**. Journal of Supercomputing, 9, 391-412, 1995.

[8] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. **Cache Performance of the SPEC92 Benchmark Suite**. IEEE Micro 13:4, 17-27. August 1993.

[9] A. J. Goldberg and J. L. Hennessy. **Performance Debugging Shared Memory Multiprocessor Programs with MTOOL**. Supercomputing'91, 481-490. Nov. 1991.

[10] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. **Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors**. Proceedings of the 23rd International Symposium on Computer Architecture, 260-270. May 1996.

<ftp://www-flash.stanford.edu/pub/flash/informLoad.ps.Z>

[11] Doug Hunt. **Advanced Performance Features of the 64-bit PA_8000**. COMPCON'95, March 1995. http://www.convex.com/tech_cache/technical.html

[12] J. R. Larus and E. Schnarr. **EEL: Machine-Independent Executable Editing**. SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1995.

<http://www.cs.wisc.edu/~larus/eel.html>

[13] A. R. Lebeck and D. A. Wood. **Cache Profiling and the SPEC Benchmarks: A Case Study**. Computer 27:10, 15-26, October 1994. <http://www.cs.wisc.edu/~alvy/papers/cprof.ps>

[14] Jussi Mäki. **POWER2 Hardware Performance Monitor Tools**. Nov. 1995.

<http://www.csc.fi/~jmaki/rs2hpm-paper>

[15] Margaret Martonosi, Douglas W. Clark and Malena Mesarina. **The SHRIMP Performance Monitor: Design and Applications**. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, May 1996.

[16] Terje Mathisen. **Pentium Secrets**. Byte Magazine, July 1994, 191-192.

<http://green.kaist.ac.kr/jwhahn/art3.htm>

[17] Alan Mink, Robert J. Carpenter, George Nacht, and John Roberts. **Multiprocessor Performance Measurement Instrumentation**. IEEE Computer, 63-75, Sept 1990.

<http://cmr.ncsl.nist.gov/multikron/articles/postscripts/mpmi.ps>

- [18] MIPS Technologies Inc. **R10000 Microprocessor Technical Brief**. October 1994. <http://www.mips.com/r10k/>
- [19] MIPS Technologies Inc. **The R10000 Superscalar Microprocessor**. Hotchips 1995. Presentation available at <http://www.mips.com/r10k/>
- [20] MIPS Technologies Inc. **R10000 Microprocessor User's Manual-Version 1.1**, Section 14.20: Coprocessor 0 Performance Counter Registers. April 1996. <http://www.mips.com/r10k/>
- [21] Silicon Graphics Inc. **Power Challenge Technical Report**. <http://www.sgi.com/Products/software/PDF/pwr-chlg/>
- [22] Silicon Graphics Inc. **Performance Co-Pilot User's and Administrator's Guide**. Document Number 007-2614-001. <http://www.sgi.com/Technology/TechPubs/>
- [23] A. Singhal and A. J. Goldberg. **Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000**. Proceedings 21st Annual International Symposium on Computer Architecture, 48-59, April 1994.
- [24] J. Torrellas, A. Gupta, and J. Hennessy. **Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System**. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, October 12-15, SIGPLAN Notices 27:9 (September), 162-174, 1992.
- [25] Jack E. Veenstra and Robert J. Fowler. **MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors**. Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 201-207. Jan 1994. Related documentation available at <http://reality.sgi.com/veenstra/>
- [26] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks. **POWER2 Performance Monitor**. PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, IBM Corporation, SA23-2737, pp. 55-63. <http://www.austin.ibm.com/tech/monitor.html>
- [27] E. Williams, C. T. Myers, and R. Koskela. **The Characterization of Two Scientific Workloads Using the CRAY X-MP Performance Monitor**. Supercomputing '90, 142-152. Nov. 1990.
- [28] Emmett Witchel and Mendel Rosenblum. **Embra: Fast and Flexible Machine Simulation**. Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, 68-79, May 1996. <http://www-flash.stanford.edu/SimOS/papers.html>

Author Biographies

Marco Zagha (marcoz@sgi.com) is a performance analyst in the Advanced Systems Division at Silicon Graphics. Previously, he was at Carnegie Mellon pursuing a Ph.D. in computer science. In fact, he better defend his thesis before SC'96 or his advisor might be pursuing him.

Brond Larson (brond@sgi.com) joined the Advanced Systems Division at SGI as a performance analyst

in 1994. Previously a senior scientist at Thinking Machines Corp., he holds a Ph.D. in theoretical condensed matter physics from Harvard University. Research contributions have focused on computational electronic structure and atomic dynamics.

Steve Turner (swt@sgi.com) is a member of the R10000 design team in the MIPS Technology division of Silicon Graphics. He received a Ph.D. in computer science from the University of Illinois in 1995.

Marty Itzkowitz (martyi@sgi.com) is the technical leader of the CASE performance tools group in the MIPS Technology division of Silicon Graphics. He has a Ph.D. in Chemistry and Physics from CalTech.