Kernel instrumentation tools and techniques

Kernel Instrumentation Tools and Techniques

J. Bradley Chen
and
Alan Eustace

TR-26-95

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Kernel Instrumentation Tools and Techniques

J. Bradley Chen
Harvard University
Division of Applied Sciences
29 Oxford Street
Cambridge, Massachusetts 02138
bchen@eecs.harvard.edu

Alan Eustace
Digital Equipment Corporation
Western Research Laboratory
250 University Avenue
Palo Alto, California 94301
eustace@wrl.dec.com

November 3, 1995

Abstract

Atom is a powerful platform for the implementation of profiling, debugging and simulation tools. Kernel support in ATOM makes it possible to implement similar tools for the Digital UNIX kernel. We describe four non-trivial Atom kernel tools which demonstrate the support provided in Atom for kernel work as well as the range of application of Atom kernel tools. We go on to discuss some techniques that are generally useful when using Atom with the kernel. Prior techniques restrict kernel measurements to the domain of exotic systems research. We hope Atom technology will make kernel instrumentation and measurement practical for a much larger community of researchers.

1 Introduction

The paper describes how the ATOM tool-building system [3, 6] can be used to build instrumented Digital UNIX kernels. Current tools for debugging operating systems and measuring their performance are subject to serious limitations. UNIX utilities such at `vmstat`, `iostat`, `top`, `kdbx` can be useful for detecting performance problems, but they are limited in terms of flexibility and the level of detail for information they can provide. Ad-hoc techniques permit more focused measurements, but require a great deal of effort and expertise to execute. The ATOM tool-building system makes it easier to create powerful tools for kernel debugging and measurement.

This paper presents four ATOM-based kernel tools which demonstrate the range of uses for ATOM kernel tools. `ksample` is a multi-processing debugging tool. `kgprof` is a gprof-style profiling tool for the kernel. Unlike traditional gprof implementations, based on statistical measures of program activity, `kgprof` gives a deterministic measure of overhead, using the Alpha cycle counter. `k3rd` is a tool for detecting kernel bugs related to dynamic memory allocation. `BT` is a branch prediction simulation, suitable for combined user/system experiments and multitasking experiments. These tools demonstrate both fundamental mechanisms for kernel instrumentation, and the broad range of problems to which ATOM can be applied. Following the tool discussion, we go on to present a number of general techniques useful for kernel tool construction.

The next section describes the additional support required in ATOM to permit kernel instrumentation. Section 3 discusses the four example tools. In Section 4 we talk about general techniques that are useful when building kernel tools. Section 5 concludes.

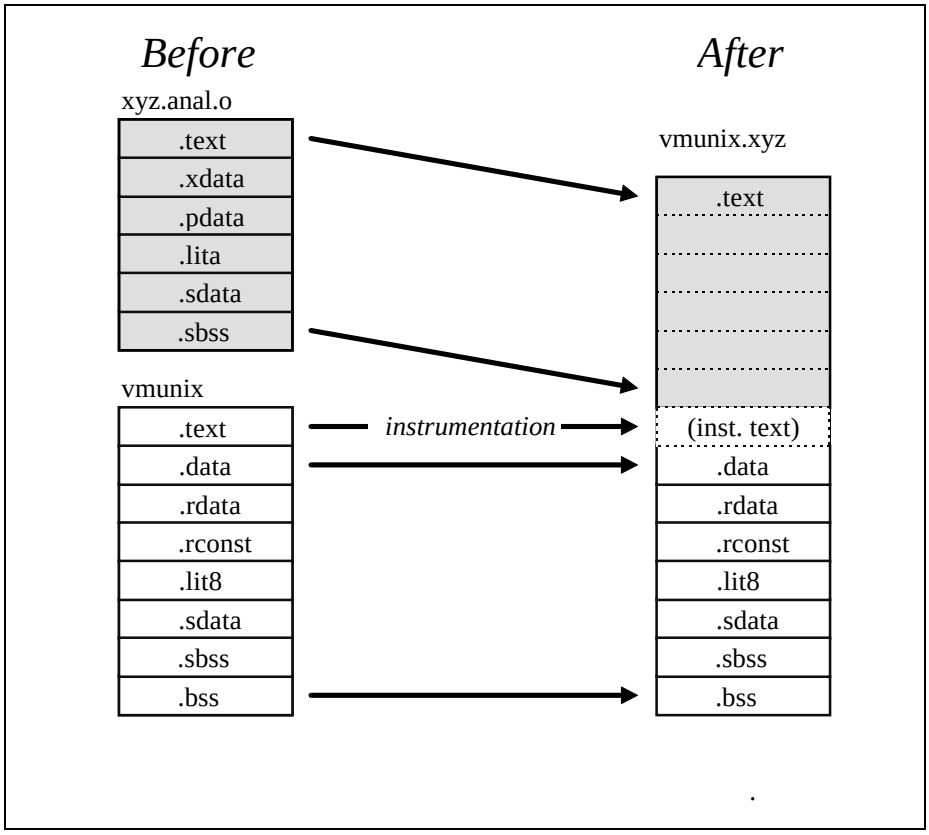

Figure 1: Instrumented Kernel Organization

2 Atom support for kernel tools

An Atom tool is implemented as two separate modules. The *instrumentation module* is a collection of C routines that specify how to instrument the program. They specify both where to put instrumentation points ("all calls to malloc", "all conditional branches") and how to instrument them ("call the routine CondBr() with arguments CurrentPC, taken or not taken, etc.). The *analysis module* implements the run-time component of the Atom tool.

Several straight-forward modifications were made to Atom to enable kernel instrumentation. The operating system kernel uses a different executable format than most executables. Because Atom reads and writes executables, it had to be adapted to perform this operation for kernel executable format.

Atom instrumentation requires taking two programs, a kernel analysis module and a program to be instrumented, and merging them into one. Although each program maintains an independent name-space, they will share a single address space. In this way the Atom tool can see the memory contents of the program it is monitoring. Figure 1 illustrates how Atom organizes the merged Digital UNIX kernel and the kernel analysis module, with all components of the tool merged into the beginning of the program text. This transformation assumes that there is room above the original text segment to load the analysis tool. To accommodate this assumption, kernels to be instrumented with Atom must be loaded with a modified text base address. This can be done by updating the value of `ALPHA_TEXTBASE` in the kernel Makefile.

Ideally standard user-level ATOM tools could be used to analyze the kernel. Unfortunately, there are several problems when working in the kernel that make this impractical:

- Some kernel procedures must not be instrumented. The instrumentation module must know about problem areas and avoid them.
- The system call interface cannot be used from within kernel analysis modules. This makes many common operations difficult, such as reading and writing files.
- `ProgramAfter` is not useful for the operating system kernel.

- Floating point computation is prohibited.
- The kernel model for dynamically allocated memory is not consistent with the model used at user level.

In practice, these restrictions are not difficult to overcome. They do require a slightly different approach to building and debugging tools.

Although Atom works correctly for the vast majority of code in the Digital UNIX kernel, there are a small number of procedures that can't be instrumented automatically and that should be avoided by Atom-based tools. Atom uses the stack to save and restore machine state at instrumentation points. Code sequences such as exception handlers and context switch code, where the stack pointer is sometimes invalid, will cause problems for Atom. Load-locked/store-conditional sequences can also cause problems, because these sequences must not include branches. If an Atom instrumentation point is placed between a load-locked and a store-conditional sequence, the branch will clear the associated flag, and the store-conditional will always fail.

The implicit goal of techniques like kernel instrumentation is to get more complete information about system activity. PAL code [5] is not a part of the operating system, and is not suitable for instrumentation with Atom, but it is something that needs to be considered if complete measurement is desired. A PAL code instrumentation system has been developed by David Mazieres at Digital Western Research Laboratory, and makes it possible to explore behavior for functionality implemented by PAL routines.

As the kernel does not begin and end in the sense of a traditional user program, initialization of analysis data structures and reporting of results must be done while the kernel is still running. One simple way to accomplish this is to map the kernel memory holding data structures of interest into the address space of a user process, so that they can be manipulated in a straight-forward way.

In this section we've given a high-level description of how Atom works. This section is not meant to document how to use Atom on a kernel. This information is provided in a separate document. [2]

3 Kernel Tools

3.1 Ksample

In multiprocessor environments, it is often difficult to understand thread interactions. This is especially true in deadlock situations, since the cause of the deadlock is often a non-trivial interaction between asynchronous system activities. Ksample is a debugging aid for multiprocessor workloads. It provides an execution history which shows thread/processor/procedure assignments, as well as a record of the concurrency that occurs during the execution of a parallel program.

Ksample instruments the beginning of each procedure in the kernel with a call to a procedure called ProcEntry. ProcEntry takes two arguments. The first is the procedure index, and the second is the value of the stack pointer. Within ProcEntry, the stack pointer is used to locate the `uthread` data structure, from which the process and thread identifier can be determined. The `whoami` PAL call is used to determine the CPU number for the calling process. A similar procedure call ProcExit is added before the exits of each kernel procedure. Each call to ProcEntry and ProcExit generates an entry in a trace buffer, which is kept in a statically declared data structure. The contents of this buffer can be examined by a user process by mapping the appropriate range of /dev/kmem into its address space. An example of Ksample output is shown below.

```
CPU 0                   CPU 1
* pid 506 *             * pid 0 *
...                     ...
=> syscall
                        <= pmap_activate
                        => switch_context
=> close
                        * pid 267 *
                        <= switch_context
                        => thread_continue
=> closef
                        <= thread_continue
```

```
=> vn_close
=> nfs3_close
                        <= thread_block
=> waitforio
                        <= mpsleep
                        <= ku_revfrom
=> sync_vp
```

Ksample demonstrates that software instrumentation tools can be useful with multiprocessor workloads. It provides an example of some of the problems with writing ATOM tools for multiprocessors, and how they can be overcome. Entries in the trace buffer need to be correctly interleaved. As multiple processors are simultaneously trying to update the trace buffer, synchronization of these accesses is crucial. We describe techniques that can be applied to these synchronization problems in Section 4. The instrumentation overhead combined with the data synchronization costs are substantial. Instrumented Ksample kernels are slowed down by a factor of 10 over uninstrumented kernels.

3.2 K3rd

Some of the most difficult programming errors to find are due to accesses to uninitialized or invalid memory. Recently tools like Purify and Third Degree improve software reliability by monitoring load and store operations and reporting accesses to invalid memory or reads of uninitialized memory. K3rd does similar analysis for the Digital UNIX kernel.

For K3rd, the kernel is instrumented to capture all instances where memory is allocated or freed by the kernel. Using these instrumentation points, the tool can identify all uninitialized and deallocated memory locations associated with dynamic memory management. These memory locations are initialized with a special magic number. Then, for each load in the kernel, the memory contents are checked against the magic number. Any load of the magic number is flagged as a possible error. A user level program can read the error log from /dev/kmem and report any errors that have occurred. A typical report is shown below.

```
1 0xfffffc000173d648 lock_read_to_write  src/kernel/kern/lock.c 737
2 0xfffffc0001719794 makealias           src/kernel/vfs/spec_vnops.c 654
3 0xfffffc000171c940 spec_swap           src/kernel/vfs/spec_vnops.c 2533
4 0xfffffc000166e230 vm_swapon           src/kernel/vm/vm_swap.c 470
5 0xfffffc0001673558 procdup             src/kernel/vm/vm_unix.c 537
```

The error report shows the address of the instruction which generated the error, the procedure name, and its location (file name and line number) in the kernel source. Although K3rd is a simple tool, it has been effective at finding a variety of subtle errors in the Digital UNIX kernel. The tool is far from perfect. First, it reports a variety of "false" errors. Most of these are caused by read-modify-write sequences. For example, most versions of Alpha do not support byte read and write operations. Thus, a single byte write is transformed into a word read, a mask operation, logical operation, and a store instruction. K3rd can report an error on the first read operation. Fortunately, most of these instruction sequences are easy to recognize, and K3rd can filter its report to exclude these errors.

A more difficult problem is the copying of uninitialized data. This often occurs when structures are copied that contain some valid and some invalid data. For example, a data structure that contains a union might be passed by value to a subroutine. This structure is copied to the stack. The read operation implementing this copy will report an error. A more elegant approach is taken in the user version of the Third Degree tool. It reports errors in the use of a data structure, where copies are not considered uses. Such refinements have not been added to the K3rd tool.

Another significant problems in implementing K3rd is that memory is allocated and deallocated in hundreds of different ways in the kernel. In contrast, most user code relies on malloc and free. Currently, K3rd only instruments a small percentage of the possible kernel allocation and deallocation procedures. Future versions will instrument many more cases, such as adding pages to the free list and allocating mbuf data structures.

Instrumention of the majority of loads in the kernel causes substantial decreases in kernel performance. Instrumented K3rd kernels typically run a factor of 15 slower than uninstrumented kernels. Although K3rd does

have an effect on dynamic kernel execution, this is less of a concern with a debugging tool than it would be with a profiling tool or a trace-driven simulation. Stressing the kernel in this way can also make it easier to find a certain class of bugs.

3.3 Kgprof

Measuring kernel performance is often difficult. Most kernel profiling tools rely on hardware clocks to sample the program counters of both the user and kernel. The values of these program counters are in turn placed in a file and are analyzed with tools like `prof`. This approach has many pitfalls. First, the hardware clock interrupt is often used as the sampling point. Since the kernel also uses the hard clock for scheduling decisions, short running threads will not be sampled. The second problem is that the duration of the test is limited by the the size of the sampling data file. Even moderate length performance tests create very large sampling data files, which must be post-processed to obtain profile information. Other problems are equally difficult to address. Sampling does not provide exact information about procedure entries and exits.

The most difficult problem with sampling techniques is that they often result in "flat" profiles. For example, they might reveal that bzero is taking up 20% of the total execution time, but fail to identify which calls to bzero are responsible. Some systems attempt to predict this information using static call graph information. This approach is not useful with the kernel, which makes extensive use of indirect procedure calls.

To address these problems, the kgprof tool instruments the start and end of each kernel procedure. Kgprof analysis procedures keep track of the kernel stack, measure time using the Alpha cycle counter, and attribute time to procedure caller-callee pairs. Kgprof provides a report in gprof style, as shown below.

```
                          callTime     called/total         parents
 %time           self    descendents     called+0          *name
                          callTime     called/total         children
-------------------------------------------------------------------------------
 35.92                   2308670051      82515/82515         INTR/TRAP
 35.92      140928541    2167741510      82515+0           *syscall
 18.93                   1216765459      40304/165078        read
 13.52                    868603956      40269/165078        write
  0.40                     25729345      82506/165078        checksig
  ...                           ...      ...                 ...
-------------------------------------------------------------------------------
 18.43                   1184489597      40304/80621         read
 13.07                    839719192      40269/80621         write
  0.01                       722164         48/80621         writev
 31.51      145562625    1879368328      80621+0           *rwuio
 16.85                   1083168141      40094/161243        vn_read
 11.58                    744418147      40100/161243        vn_write
  0.66                     42564834      80622/161243        getf
  0.11                      6879371        217/161243        soo_write
  0.04                      2337835        210/161243        soo_read
  ...                           ...             ...         ...
```

Kgprof introduces another set of interesting problems. First, the data on callee-caller pairs is kept in data structures that are shared among all the kernel threads. When an interrupt or thread switch occurs in the kernel, kgprof must know not to attribute time consumed by other kernel threads to the thread that has been suspended. In Section 4 we discuss the techniques for dealing with asynchrony in the kernel.

3.4 BT - Branch Prediction Simulation

We used Atom to implement a simulator of branch prediction hardware. The tool allows simulation of a variety of branch prediction strategies and can measure kernel only, user only, or kernel plus user activity. Multi-tasking workloads are also possible.

BT uses instrumentation points at conditional branches for branch count and branches taken. The data collected at these points drives a simulation of various dynamic branch prediction strategies. Instrumentation points on each basic block are also used to get a total instruction count for the workload. An optional flat profile of procedure calls was implemented for monitoring and debugging the simulator.

BT uses data structures allocated in kernel memory and mapped into user memory to allow the kernel and multiple user workloads to share the same data structures and participate in the same simulation. We use optimistic concurrency control to prevent asynchronous activities from corrupting the shared data structures. BT uses load-locked and store-conditional sequences to protect atomic updates of shared data. If, at the end of a atomic sequence, it is found that the sequence has been interrupted, a failure is recorded and the update is aborted. This gives reasonable results provided that the number of interrupted sequences during a simulation is small relative to the total number of events. As interrupted atomic sequences are rare, this gives good results with minimal overhead for concurrency control. With this technique we also avoid the possible ill effects of manipulating the hardware interrupt level.

BT slows program execution by as much as a factor of 15. It avoids distortion from time dilation by scaling clock interrupts in software. We discuss this technique in detail in Section IV.

The four tools discussed in this section give an idea of the. range of applications for Atom kernel tools. In the next section we present some general techniques that are useful when building tools of similar complexity.

4 Techniques

4.1 Debugging Instrumented Kernels

Incorrectly instrumented kernels are almost impossible to debug. The best strategy is to not make mistakes. Start with implementations that are known to work, and make small incremental changes. When possible, test instrumentation and analysis modules at user level before implementing them in a kernel-level tool.

Kernels usually fail very early in the boot process. This is good, because the bad kernel doesn't have a chance to significantly alter the state of the machine. We have booted (or attempted to boot) hundreds of bad kernels, but have never had to rebuild disks or reinstall software.

Some mistakes are trivial to diagnose and fix. One error message that occasionally appears at boot time is shown below.

```
>>> boot -fl i
INIT-S-CPU...
INIT-S-RESET_TC...
INIT-S-ASIC...
INIT-S-MEM...
Enter [kernel_name] [option_1 ... option_n]: vmunix.crash
Loading vmunix.cBootstrap address collision, image loading aborted
?05 HLT INSTR
  PC= 00000000.20010610 PSL= 00000000.00001F00
>>>
```

This message indicates that the `ALPHA_TEXTBASE` has not been set to a large enough value. Recall (Section 2) that ATOM adds code and data for the kernel tool to the beginning of the kernel text segment. This memory space is reserved statically by modifying the kernel makefile to shift the kernel text base address, `ALPHA_TEXTBASE`. If the reserved space is not adequate, the memory for the ATOM tool will overlap with memory reserved for the kernel bootstrap program.

Other error messages are not so easy to decipher. The most common error message is shown below. This is the equivalent of the "segmentation fault, core dumped" message for user programs.

```
>>> boot -fl i
INIT-S-CPU...
...
OSF boot - Wed Nov 16 00:10:16 EST 1994
Enter [kernel_name] [option_1 ... option_n]: vmunix.bug
Loading vmunix.bug ...
Current PAL Revision <0x1052f>
Switching to OSF PALcode Succeeded
New PAL Revision <0x20122>
Loading into KSEG Address Space
Sizes:
text = 5635888
```

```
data = 1146336
bss  = 2371152
Starting at 0xfffffc0003f27350
?02 KSP INVAL
  PC= 00000000.000604E5 PSL= 00000000.00000007
```

If you are very lucky, the PC will correspond to a known address in the kernel. A kernel virtual address can be converted to a physical address by ANDing with 0xffffffff. The corresponding procedure can be found using symbol table information for the instrumented kernel. Unfortunately, it is often the case that the error PC does not correspond to an address in the instrumented kernel. In these cases it is sometimes useful to use console commands to examine the contents of physical memory. Memory contents can be used along with symbol table information to examine the state of the kernel and of data structures in the kernel analysis tool at the point of failure. Many difficult bugs have been found with the aid of data structures included specifically for debugging. For example, a circular buffer can be used to record the sequence of events (such as procedure entries and exits) immediately preceding a system crash.

The examination of memory contents after a kernel crash may seem like a very primitive technique for solving complex debugging problems, but this is purely a question of point of view. For some bugs, memory contents are erased after a failure. In these situations, an in-memory trace can seem like a great luxury and a very advanced technique. Debugging remains one of the challenges when writing complex Atom kernel tools, and at present there are no debugging tools which can substitute for careful programming and experience.

4.2 Dealing with time dilation

Time dilation occurs because operating system kernels instrumented with ATOM execute more slowly than the standard kernel. These slowdowns can be significant. Table 1 gives the slowdowns for kgprof, BT, and a kernel version of the Pixie [4].

kgprof	5.4
kpixie	7.0
BT	15

Table 1. Atom Tool Slowdowns.

Slowdowns for tools that place a significant number of instrumentation points tend to be between a factor of four and a factor of 100. This slowdown affects idle time and events generated by background activity such as clock interrupts, system daemons and periodic network routing table updates.

Time dilation distorts the latency of I/O operations. Instructions appear to execute slowly but the speed of I/O devices remains the same, so the I/O devices appear to be many times faster than they really are. For workloads with a single thread or process and synchronous I/O, all I/O time is spent in the idle loop. Under the assumption that no useful work is done during idle time, idle loop activity can be ignored and the actual amount of idle time becomes unimportant.

Non-trivial idle-loop activity has the potential to complicate this situation. In Digital UNIX, significant idle time is spent in a routine called **vm_page_tester** which scans the address space for valid physical pages. Empirically it is difficult to see how this activity would have any significant effect on program execution time. In our experiments, we have not observed a workload for which this activity has significant impact. As part of a more comprehensive solution to this time dilation problem, we are studying the problem of scaling idle time.

Unlike I/O activity, background events are always asynchronous and are independent of the workload in question. While apparent idle time decreases for instrumented systems, background activity appears to increase. Without compensation, background events would dominate the measured activity, and the distortion to experimental results would be excessive.

As a simple example, suppose that background activity is responsible for 1% of total execution time. Assuming a 100 second compute-bound workload, background activity would account for one second, with the other 99 seconds going to the main computation. An Atom tool that slows instruction execution by a factor of 50 will increase the wall-clock time required by the main computation to 4950 seconds. Background events will occur with the same frequency, but they will take 50 times longer to service, requiring 50 * 1% or 50% of the total

execution time. As a result the simulator will see 99 seconds of useful work, due to the workload, and 9900 seconds worth of background activity. Without compensation, the apparent proportion of background activity goes from 1% to 99%.

Background activity occurs in response to either internal or external events. External events are from things like periodic network packets. Scaling activity external to the experimental system requires the use of a dedicated network, with background scaled or eliminated. Internal events are triggered by clock interrupts or other timers. One way to scale this activity is to re-configure the timer chip to change the clock interrupt rate [1]. A drawback of this technique is that the available clock interrupt rates are limited by the hardware. When there is not a good match between the slowdown factor for a given tool and available clock interrupt rates, distortion will result. As the slowdown for ATOM tools is highly variable, a good match between tool slowdown and hardware clock interrupt rate is not always available.

An alternative to scaling clock interrupts in hardware is to scale them in software. This can be achieved by replacing the the call to the clock interrupt routine with a stub routine that calls the real interrupt routine only a fraction of the time, scaling the number of calls according to the tool slowdown. This technique makes it possible to implement an arbitrary clock scaling factor. It has the nice side effect that all time-related quantities in the kernel (such as scheduler quantum) are scaled appropriately.

A third alternative is to ignore activity generated by clock interrupts by not instrumenting it, or by disabling instrumentation code during a clock interrupt. This approach works for synchronous activity. Unfortunately, a significant part of clock interrupt activity in Digital UNIX is asynchronous. Threads are scheduled by the clock interrupt handler, with the real work occurring after the clock interrupt handler has returned. Disabling instrumentation code is not possible because the clock interrupt event has terminated. However, this activity shares much common code with other parts of the system, so it must be instrumented. For these reasons, completely ignoring clock interrupt activity is not a satisfactory solution for Digital UNIX.

Note that, for experiments restricted to synchronous user-level activity, time dilation has no impact on the behavior seen by the simulation. Multi-tasking workloads do require compensation, so that I/O time and scheduler activity will scale appropriately. Similarly, when only the kernel is instrumented, as is the case with kgprof, clock interrupts in user mode should occur at the normal frequency.

4.3 Dealing with asynchrony in the kernel

Kernels create and destroy threads frequently. Each threads can be interrupted at arbitrary times by threads of a higher priority. For multi-processors, several kernel threads can be running simultaneously. Tool developers must be keenly aware of these problems to avoid synchronization problems.

As an example, consider an ATOM tool developed to count the number of calls to the `bcopy` procedure. A naive approach would be to add a call to the Count analysis procedure before the start of the bcopy kernel procedure. The Count analysis procedure would load the count variable into a register, add one to it, and write the result back to memory.

Unfortunately, if the analysis procedure is interrupted after the read of the count variable, and a new thread executes the bcopy procedure, the old value of the count variable will be incremented. When the original thread continues execution, it writes an incorrect result. A simple approach to avoiding this problem is to keep an array of counts, one for each interrupt level. Since threads at the same interrupt priority level cannot interrupt each other, the counts are updated correctly.

Now assume a similar Atom tool running on a multiprocessor. In this case, multiple concurrent processes can update the same variable at the same interrupt priority level. The only complete solution is to lock the variable before the update. Unfortunately, acquiring and releasing multiprocessor locks is relatively expensive.

A further approach to avoiding collisions is to eliminate conflicts altogether. For example, a simulator that includes both user and kernel activity might have a counter for instruction cache misses. Conflicts between kernel and user can be avoided by counting user and kernel cache misses separately. Similarly, data structures can sometimes be allocated on a per-thread basis to avoid conflicts between kernel threads. In some cases a tool can be written such that, even though conflicts are not avoided altogether, the probability of a conflict is small. This can be done by making the critical section very short (as in BT) or by spreading counters out over many unique data structures (as in kpixie). When conflicts are infrequent, it becomes possible to use more expensive

mechanisms to recover from them, or (in some cases) to simply keep a failure count and abort the actual update. For many tools, aborted updates become irrelevant when they are sufficiently infrequent.

4.4 Conclusion

Atom kernel tools are not hard to write. Our experience demonstrates that instrumentation techniques are useful for a broad range of applications, extending well beyond the profiling and simulation tools created with previous instrumentation systems. Although some new problems arise when using instrumentation tools with the kernel, they can be overcome. With Atom kernel tools, kernel measurement can become a standard tool for the computer systems research community.

References

[1] J. Bradley Chen, Anita Borg, and David Wall. "Software Methods for System Address Tracing: Implementation and Validation," Digital Western Research Laboratory Technical Report 94/6, September 1994.

[2] Alan Eustace and J. Bradley Chen, *Atom Kernel Instrumentation Guide*, August 1995. Available from the authors.

[3] Alan Eustace and Amitabh Srivastava. ATOM: A flexible interface for building high performance program analysis tools". *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, pp 303-314, January, 1995.

[4] MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.

[5] Richard L. Sites, Editor. *Alpha Architecture Reference Manual*, Digital Press, Burlington MA, 1982.

[6] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *Proc. of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.