A kernel trace device for Plan9[*,†]

Ronald G. Minnich
John Floren
Aki Nyrhinen

May, 2008

Abstract

We describe a Plan 9 trace device, devtrace, its uses and its implementations. The trace device can be used to selectively trace functions and processes in Plan 9. Users can enable a range of functions to be traced, observe which of the functions are called, in what order, what their parameters are, and the time spent (in CPU ticks) in each function. We have devloped a set of tools for plotting this data to make the progression and timing of function calls clear. Since all Plan 9 file systems are user level processes, it is possible to trace a single process file I/O as it progresses from the process, through the file server processes, and to disk. This measurement, in turn, allows us to propose changes in the Plan 9 kernel design and implementation to improve performance.

The implementation of the trace device went through several distinct phases. In the end, we arrived at a device with a textual interface. Users need not write programs to use the trace facility. The trace device does not rewrite kernel code and hence does not require priveleged access (as in Linux or Solaris). Any user of a Plan 9 terminal can measure their system's performance.

The trace device was designed to help us with performance evaluation of Plan 9 on two supercomputers, the Cray XT4 and the IBM BG/P.

1 Introduction

This project started out as a simple question: where is the time going in Plan 9, and why? Plan 9, in earlier years, had been uniformly faster than other operating systems. In recent years, however, the continuing improvement in competititors, particularly Linux, has begun to show. Plan 9 loses badly on microbenchmarks such as the lmbench pipe throughput test. Interestingly enough,

[*] Sandia National Laboratories

[†] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DEAC0494AL85000.

Plan 9 continues to “feel” faster on slower hardware, showing that microbenchmarks are not everything. But, nonetheless, we wanted to be able to isolate the overheads in Plan 9 and see if either an implementation or architecture change was appropriate. Our primary use of Plan 9 is in High Performance Computing (HPC) systems, in which overall throughput can depend on very small overheads that don’t much matter in desktop systems. Isolating problem overheads and removing them is a very common activity in HPC.

Plan 9 is a very small, tightly crafted operating system. It has only 40 or so system calls. A given file system IO call will result in a call stack that is only a few levels deep, as opposed to the (literally) dozens of call levels found in, e.g., a Linux NFS I/O. Modifying Plan 9 system calls in simple ways is a far less daunting task than on other operating systems. Also, Plan 9 is very modular; the boundaries between components are well-defined and adhered to, with very little of the shared state that characterises most operating systems. This separation enables the inclusion of changes as long as they do not break the interfaces. Hence, it is very likely that, given the discovery of a major overhead that might be avoided by a straightforward redesign, the redesign can be incorporated in the kernel. Recently, the virtual memory subsystem was almost completely redesigned and it affected only one other source file – the early bootstrap code.

To give some flavor of what the trace device allows, we show a real trace in Figure 1. The data and plot were created using devtrace and a processing pipeline described later in this paper. The trace shows the kernel functions called by an ’echo’ command. Echo writes its arguments standard out using the write system call. As expected, most of the time is spent in the write system call itself, and a huge fraction of that time is spent in memmove. It can be seen that the memmove in this case is bracketed in time by the write system call. Since we are only tracking this one process, we can attribute all of the time spent in memmove to that system call. This overhead raises all sorts of questions, which can for the most part be answered with devtrace.

As with most Plan 9 devices, the interface to devtrace is textual. There are two files: tracectl and trace. The tracectl file is used to both query the trace device and control its actions; the trace file is used to read trace records. To provide some feel for how these files are used, we show a sample trace interaction in Figure 2. To see the initial state, we start by cat’ing the tracectl file with no tracing set up; it shows the size of the trace log buffer (8192 trace records); and the process IDs (PIDs) that are being watched (watch 0, meaning all PIDs) along with some internal information, the most important being the tracehits (how many traces the device has recorded) and the number of records in the queue (’in queue’).

The trace device is not set up to do anything; to use it, we next set up a trace. We first create the trace description, specifying a start and end program counter (PC) range. For convenience, we allow the addresses to be specified as an offset from the start of kernel text (we can specify the full address but this form is a lot easier to type). Every trace description has a name. In this case we are tracing all the functions in the real time clock (rtc) driver so we call this

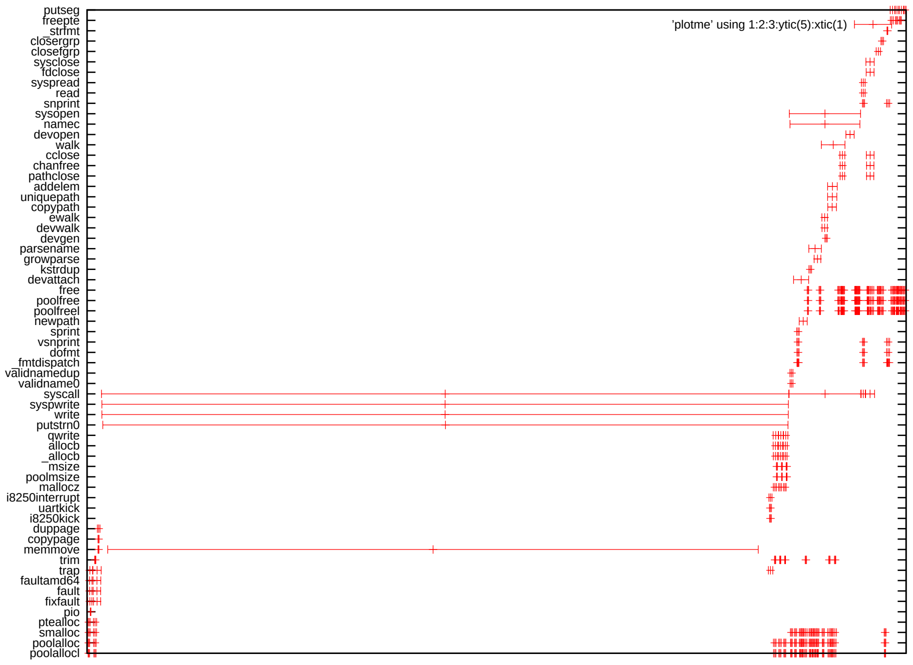

Figure 1: A sample trace output with tracedev. The X axis is in units of processor ticks.

```
cpu% cat /dev/tracectl
logsize 8192
#tracehits 0, in queue 0
#tracelog ffffffff81a48d68
#newplfail 0
#traceactive 0
#slothits 0
#traceinhits 0
watch 0
cpu% echo trace 119f61 11aac8 new rtc > /dev/tracectl
cpu% echo trace rtc on > /dev/tracectl
cpu% echo start > /dev/tracectl
cpu% cat /dev/rtc
1212487238
cpu% cat /dev/tracectl
logsize 8192
trace ffffffff80119f61 ffffffff8011aac8 new rtc
#trace ffffffff80119f61 traced? ffffffff815e4778
trace rtc on
#tracehits 362, in queue 362
#tracelog ffffffff81a48d68
#newplfail 0
#traceactive 1
#slothits 365
#traceinhits 181
watch 0
cpu% echo stop > /dev/tracectl
```

Figure 2: Setting up a trace session.

trace 'rtc'.

Traces are not enabled by default. To enable a trace, we issue the 'trace rtc on' command, using the name we set up in the previous command. We also have to actually enable the trace device itself, with the 'start' command.

This setup is very similar to a logic analyzer setup. We create the triggers, enable some of them, then start the logic analyzer. Once we have initiated activity to (hopefully) trigger the filters, we stop the analyzer and look at the results. In this case, running the command: `cat /dev/rtc` should cause some rtc functions to run and trigger the rtc trace description we set up earlier. To check the state of the trace device, we can cat /dev/tracectl again, and we see that there are 362 hits and 362 records in the queue. We next stop the trace device; it is time to look at the data.

As you can see, the tracectl contains information and commands; since the output of tracectl is a valid input to the tracectl, one can cat the tracectl file and save it, using it to set up the identical trace later.

```
E ffffffff8011aa68 000088297eecd2e7 000000000000023c 00000000000007be
ffffffff81accb78 ffffffff81a4d2a8 ffffffff0000000a
X ffffffff8011aac7 000088297eecd445 000000000000023c ffffffff801abc78
0000000000000000 0000000000000000 0000000000000000
```

Figure 3: Data output from /dev/trace

Figure 3shows two lines of output from the trace device. The format of the line is:

- E or X indicating entry or exit
- The PC
- The processor time stamp counter, 64 bits
- The process ID
- E records: the first four arguments[1]; X records: the return value

The rest of this paper is structured as follows: first, we provide an overview of related work; then we describe the current implementation and the post-processing tools that we have developed to analyze the traces. We next describe some possible alternatives to our implementation. Finally, we close with a description of future work.

2 Related work

There have been many implementations of kernel tracing facilities over the years; more tracing facilities than there are operating systems. Trace systems generally have one of three goals: debugging, which requires a tremendous amount of flexibility; tracing calls and times, which requires recording function invocation, arguments, and timing; and profiling, which requires only recording the function start address and time spent in that function, which is accumulated in a histogram (i.e. time for individual function calls are lost; only the total time spent in each function is recorded). There are common techniques for implementing a trace system, namely:

1. rewrite-based (a.k.a. self-modifying code). A program or driver sets a breakpoint or a jump by actually rewriting part of the kernel text. The kernel manages the breakpoint or jump by calling a specified function. The function can be pre-defined and the same for all traces, or arbitrary and specified when the trace is created.
2. code-based. Programmers insert a call to a logging function at various places in the code at compile time.

[1]If the function takes fewer than four arguments the extraneous argument values are invalid and should be ignored.

3. automated. This technique is a variation on code-based. As part of the kernel build process, the compiler or linker generates code that calls a function for each function entry and exit.

4. hardware. Code is instrumented by hardware using special-purpose attached I/O devices or logic analyzers**(author?)** 1. This technique often requires some form of code-based support to write tags to the hardware at the proper points.

2.1 Rewrite-based

Rewrite-based tracing replaces a portion of the kernel code with code that calls a handler.The trace setup code allocates a buffer, saves the written-over code in it, and installs a jump to the code buffer in place of the written-over code. The buffer contains written-over code, a call to the trace support code, and code that resumes the function. See Figure 4 for an example. A corresponding code buffer is created for function exit. The handler can be a user-provided function, sometimes called "trigger code". Typically, trigger code logs the event and variables of interest, which are almost always the parameters to the function.

The code that is written over can be replaced with a breakpoint instruction or a jump instruction. Kprobes, in the Linux kernel, is breakpoint-based; djprobes is jump-based – although as part of the installation process, djprobes begins by inserting breakpoints. Breakpoints have the advantage of being small, usually one byte; they have the disadvantage of high cost in time, since a breakpoint ends up in the interrupt handling code.

In some of these systems, a pre-written function is called; in others, users tracing kernel code must write a complete kernel module containing the trigger code functions. The trigger code can be used for more than one traced function but must be able to disambiguate the multiple callers - i.e., function exit and entry, or different functions. In Kprobes, DJprobes, and many other systems, the user must write a module that explicitly names the functions to be traced, *when the trace module is compiled.* Further, users wishing to export the data from the trigger code to user mode must write additional code to move the data to another kernel subsystem (e.g. relayfs, now known as relay), from which the user program can extract it from the kernel. Kprobes supports the tracing, but not the data transport.

While this code replacement might seem straightforward, albeit tricky, when the impact of shared memory multiprocessors, caches, interrupts, and all possible corner cases are considered, it is quite complex. Some of the issues include:

- For a period of time, as the function is being modified, *the function code is in an invalid state and will crash the machine if executed.* It is essential that the modified function code and the code buffer not be used while the rewrite is happening.

- Whether any processors are executing code that is to replaced with a jump. Here is one reason that an interrupt instruction is preferred: it is one byte,

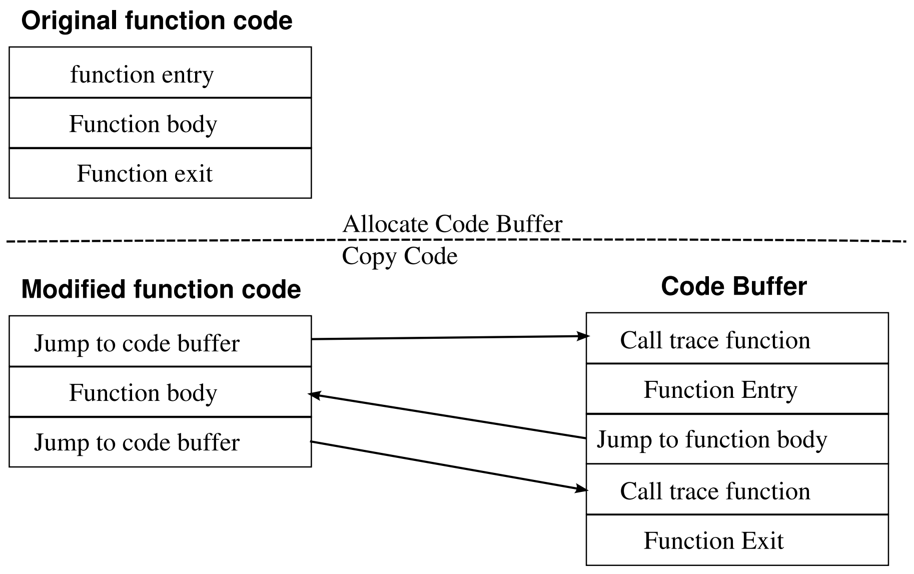

Figure 4: An example of jump-based probes

so there are no issues with multibyte instructions that might span cache lines.

- How much of the code needs to be copied. On an i386 or other CISC CPU, with variable length instructions, the jump instruction might cover 0, 1, more instructions. In fact, the jump instruction rarely covers an integral number of other machine instructions, so that although the jump instruction might be only (e.g.) 5 bytes, more than five bytes might need to be copied.

- The actual content of the written over code. The written over code might not be position independent; might contain a jump or an interrupt call; might be an interrupt handler; or it might cause a page fault. All these cases must be handled. Kernel code has to evaluate the code, and determine whether to run the code in the buffer, emulate the code in software, or return the code to its place of origin and run it in single-step mode. We show the latter scenario in Figure 5.

- Whether the handler code might itself be probed.

- Whether the code buffer can be freed. At what point can the kernel be sure that no processes are executing the code buffer? The trigger code might take a page fault, call another kernel function, or abort and never return.

- Whether multiple probes are to be attached to a single function.

- Whether the process installing the traces exits cleanly or not.
- Whether a kernel module is installing a trace on itself.

Multiprocessor machines add a host of difficulties. The code may be rewritten in one processor, but we have no guarantee of when or if other processors will see the changes – or, still worse, see some but not all of the rewritten memory. As pointed out in **(author?)** (2), the problem "... is equivalent to the problem of finding RCU quiescent state without rcu_read_lock()/rcu_read_unlock().". The task of moving code so as to redirect it requires that we determine if the execution of an arbitrary piece of code will ever terminate,and is in fact equivalent in difficulty to the halting problem.

The Dynamic Kernel Modifier (DKM) solved one part of this problem, but at a great loss in flexibility. DKM was limited to replacing only a known common set of function prologues, identified by their binary signatures. These function prologues contain only push or pop instructions, load constant into registers, or register to register moves. We determined that this set of limited instruction sequences covered a significant fraction of the function prologues in the Linux kernel, mainly because gcc uses the callee-save model and generates very similar code for many functions. DKM's simpler design and code flow were possible because the function prologue code was location-independent. DKM did not solve the SMP problem, however, and hence only eliminated a fraction of the complexity of kprobes or djprobes.

2.1.1 dtrace

This section would be incomplete without a reference to Sun's dtrace, possibly the most sophisticated rewrite-based system, and certainly the standard by which all other kernel trace tools are measured. Dtrace is a debugging oriented tool, and hence has a great deal of flexibility. Dtrace can set a probe point on any of tens of thousands of places in the Solaris kernel. The trace points can run always installed, since their cost when not activated is zero. Unlike most other Linux or Unix trace systems, dtrace provides a rich support system for naming probes, acquiring the data created when probes are triggered, and processing the data to simplify analysis. Dtrace supports both so-called "static tracing", essentially a code-based tracing mechanism, and a "function boundary" tracing mechanism, implemented with the same technique as DKM: relying on the fact that function entry and exit have a characteristic set of location-independent instructions. Dtrace replaces one instruction with a TRAP instruction, and, rather than executing the written over code, emulates it in software. Consequently, the cost of executing a dtrace trigger code is fairly high. Dtrace shares the problem of most code rewriting strategies, in that the kernel code can be in an invalid state while the rewrite is being done.

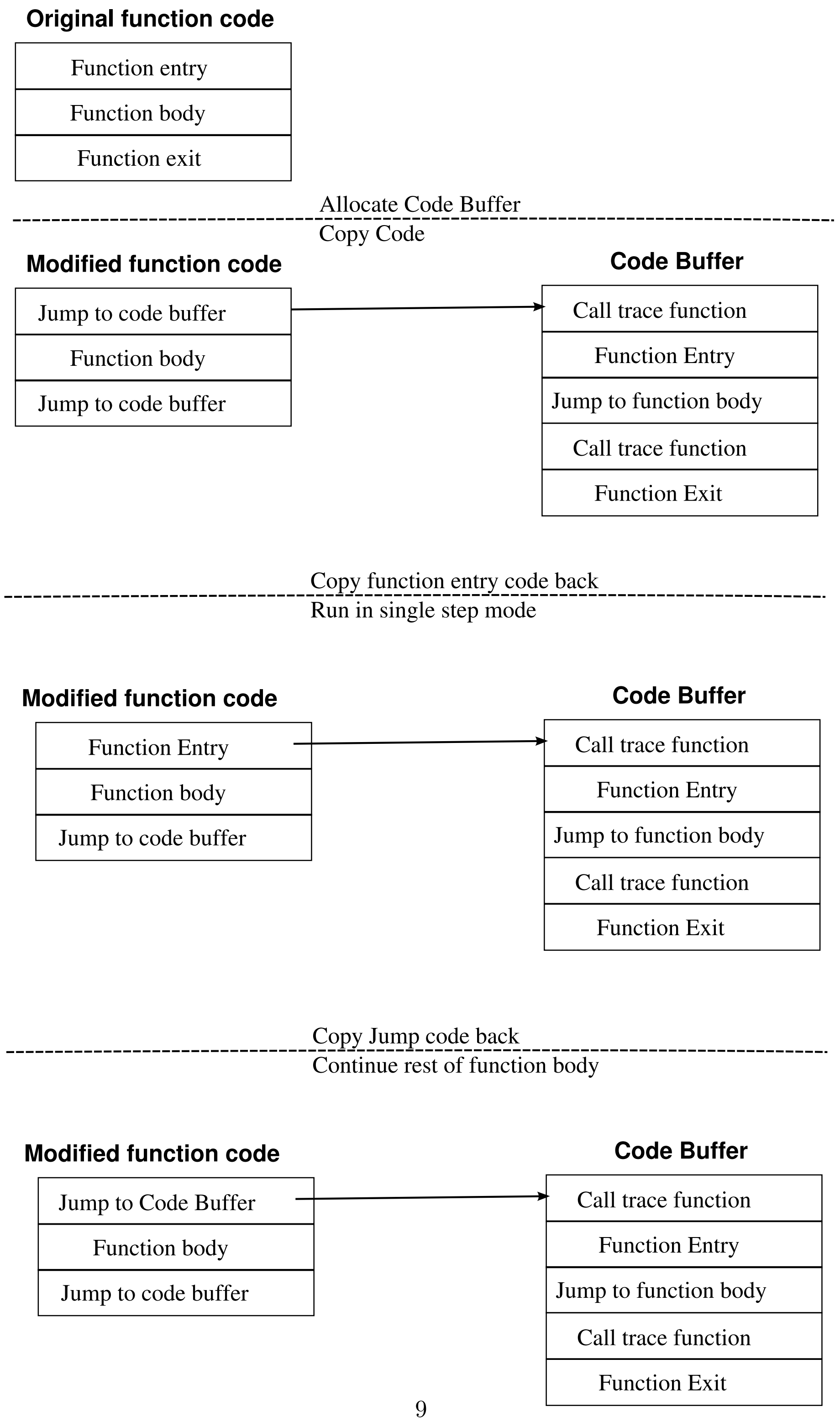

Figure 5: Modifying the code, restoring it, and running it

2.1.2 Performance issues with code rewrite tracing

As we can see, code rewriting is complex and has a number of non-obvious costs: making sure no processors are executing code that is being rewritten; making sure all processors see the changes once they are finished; executing the code that has been moved; and managing the problems that can occur when arbitrary code has been moved to another location.

One additional concern that is not immediately obvious, for performance measurement, is the *differential cost* of executing a function when it is traced. Consider the case when non-traced functions, in a tracing-enable kernel, see no performance penalty. Those functions that are traced will appear to have a much higher comparative cost than they do in reality. If we measure, e.g., the performance of a program that uses non-traced functions, we will artificially inflate the cost of a program that uses traced functions.

In contrast, in the Plan 9 trace device, the time to run all functions is uniformly increased whether they are traced or not. As a result there is a closer correspondence between the time to execute two functions, and hence two programs which use those functions, even if one is set up to be traced and one is not. In short, a zero cost penalty for non-traced functions could lead users to attribute a higher time cost to traced functions than is in fact the case.

2.2 Code-based

Code-based systems have been around for some time. One of the earliest Unix implementations could be found in the SunOS kernel. Setting up a trace required the programmer to write a line of code that looked something like this:

trace(mask, arg1, arg2, arg3, arg4);

These lines were conditionally compiled in, and the mask allowed for finer control of compiled-in trace calls.

A more recent version of code-based tracing is Linux kernel markers. Programmers insert "markers" at points of interest in the kernel source, e.g.:

trace_mark(blk_request, "count is %d", count);

The markers are disabled by default. They are enabled by calling a function which names the marker, and contains a function pointer and a pointer to private data. The function will be called when the marker is hit, with the data passed to it. In order for kernel markers to become generally useful, large parts of the kernel – all 50 Mbytes of it – need to have kernel markers added. Adding this additional code to the kernel is quite a major effort and will take some time. As of 2.6.25, only four markers have been created.

2.3 Automated trace

In an automated trace system, the kernel or the linker generates the trace support code via a build-time command. An example is the Plan 9 kernel profiling facility, which is invoked from the Plan 9 linker. The linker in Plan 9 is capable of inserting code or optimizing code away – it does far more than a traditional linker, sharing code generation responsibilities with the compiler. When it is invoked with a -p switch, the Plan 9 linker inserts a call to _profin at the entry point of the function, and a call to _profout at each exit. The profiling library is also linked in as part of this process. The only information passed to and used in the profiling functions is the program counter. The counter is used to create a histogram of time spent in functions. Relationships between functions, and time for certain types of calls, is not collected.

This facility can be modified to implement tracing, not profiling, as we discuss below. To implement automated tracing, we can rewrite the profin and profout functions.

2.4 Data extraction and analysis

As mentioned above, tracing is only part of the problem. Once the trace function has been activated, it must produce information and deliver it to a consumer. The simplest consumer is the kernel system log. Data is produced for the log by a print function. The bandwidth provided by the log, and the performance impact of using it, are such that it is rarely used: it is very easy to create so much data from printing that it is overrun and lost. Instead, the trace facility can provide a way to provide data for user-level consumers, as in dtrace; or, the trace facility might require that users set up the means by which data is delivered to consumers, as in kprobes, djprobes, and other systems.

Another issue concerns the format of the data. Kprobes, djprobes, and kernel markers all allow unrestricted creation of data streams, both in content and record size. While a lack of restrictions might seem desirable, it can be difficult for programs to parse all the possible variations of data output – this same problem has been seen and documented for, e.g., /proc**(author?)** (3). The four markers present in 2.6.25 have this format:

"name %s format %s"

"name %s format %s",

"ctx %p spu %p",

"ctx %p",

This problem has been dealt with before, and it is easy to solve: if the data size and content are arbitrary, then the format should be in a self-describing, self-contained format, e.g. s-expressions as defined in**(author?)** (3). A self-describing format has many advantages, not the least being that output from multiple sets of markers can easily be processed by a program which is only processing a subset of the markers. Programs need not concern themselves with all possible marker formats, since the self-describing structure of the data makes it easy to skip markers that are not of interest. Data can be saved and resurrected years later, and the structure of the data is readily apparent.

DKM, dtrace, and the Plan 9 trace device (devtrace) opt for a fixed-format, fixed-size data format, for reasons of processing complexity and overhead. DKM and devtrace also fix the content of the data: a function entry/exit tag; a cycle counter (processor clock); the program counter; and the first four parameters (entry) or the return value (exit) of the function. Dtrace has a bit more flexibility but also has a fixed record size. Both DKM and devtrace provide the tracing facilty and an I/O device from which to read trace events.

2.5 Summary

We have only touched upon a small fraction of the many available tracing facilities. Tracing facilities have been developed over at least the last four decades; we focus mainly on the Linux systems as they are the most likely to be familiar to the reader. The systems vary little in their implementation; some are dynamic, and installed by code rewrite; others are written into the kernel as code by programmers; still others are inserted automatically into the kernel by the compilation toolchain.

The system we have built for Plan 9 (devtrace) is based on the automatic approach. We build the kernel with profiling enabled but replace the normal profiling functions. Our trace functions allow users to conditionally enable both individual functions and individual processes, to allow us to trace (e.g.) file I/O calls from an editor to the file server and back. The control of which functions and processes to trace is accomplished by writing textual commands to a control file, in the usual Plan 9 manner. We now describe this system in more detail.

3 The Plan 9 trace device

The Plan 9 trace device is an automated trace device that does not use code rewriting. To use it, programmers add the -p switch to the Plan 9 linker command for the kernel, and also link in two additional files: the C code for the trace device itself and the assembly code that implements and replaces the standard _profin and _profout functions. The assembly code is needed to ensure that the interposed profiling calls do not interfere with argument or return values.

The _profin/_profout assembly code is limited to the minimal support needed on a per-architecture basis. The functions test to see if tracing is globally enabled and, if so, push the first four args (on entry) or the return value (on exit) and call C functions named tracein and traceout. The main modification from the standard functions is in the provision of the additional information. As an example we show the assembly support code for the AMD Opteron in Figure 6.

The C code implements the rest of the tracedev functionality and, again, provides a device interface for controlling the device, determining status, and reading the data. In the Plan 9 manner, the device supports two files: ctl and data.

Data file

The data file is read-only and, when read, returns trace records as text. For 32 bit architectures, the records are 64 bytes long and formatted as a follows:

```
E 00000000 0000000000000000 00000000 00000000 00000000 00000000
X 00000000 0000000000000000 00000000 00000000 00000000 00000000
```

```
TEXT _profin(SB), 1, $0
/* check the global trace flag */
CMPL traceactive(SB), $0
/* skip this code if tracing is not ready */
JEQ inotready
/* push arg 4 */
MOVQ 32(SP),AX
PUSHQ AX
/* push arg 3 */
MOVQ 32(SP),AX
PUSHQ AX
/* push arg 2 */
MOVQ 32(SP),AX
PUSHQ AX
/* push arg 1 – which is held in bp */
MOVQ BP,AX
PUSHQ AX
/* push the PC */
MOVQ 32(SP),AX
MOVQ AX,BP
PUSHQ AX
CALL tracein(SB)
/* pop stack */
POPQ AX
POPQ BP
POPQ AX
POPQ AX
POPQ AX
inotready:
RET
/* slightly easier: only need to save AX, the return argument */
TEXT _profout(SB), 1, $0
/* again, skip if we're not enabled yet */
CMPL traceactive(SB), $0
JEQ notready
/* save the return arg */
PUSHQ AX
MOVQ $0,AX
PUSHQ AX
/* move return arg to BP, which is arg 1*/
MOVQ 16(SP),BP
CALL traceout(SB)
POPQ AX
POPQ AX
notready:
RET
```

Figure 6: Assembly support for Opteron

```
logsize 8192
trace ffffffff80168c50 ffffffff80170000 new v
#trace ffffffff80168c50 traced? ffffffff8159fdb8
trace v on
#tracehits 0, in queue 0
#tracelog ffffffff819d7e28
#newplfail ffffffff00000000
#traceactive 1
#slothits 3
#traceinhits 0
watch 84
```

Figure 7: Sample output from the ctl file

The E or X indicates function entry or exit. The first word is the PC, the second the time stamp counter,the third the PID. E records show the first three parameters, and X records show the return value.

On 64-bit architectures, the records are 128 bytes long, as follows:

E 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000

X 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000

E records on 64-bit machines show the first four parameters.

Ctl file

The ctl file, when read, returns information about the state of the device. As in many Plan 9 devices, strings read from the ctl file contain valid commands; the output of the ctl file can be saved and written back to the ctl file. The main use of the ctl file is to control tracing. Programs, scripts, or users echo commands into the file. The commands are shown in Table 1.

This set of operations allows tracing some or all of the kernel. We can restrict the set of functions traced, as well as the set of processes traced. We can follow a write system call from a process to the file system server, its archival backup, and from there to the main disk drive. We can look at the sizes of writes and determine where the bottlenecks are in the I/O system. Most importantly, by design, the overhead is uniform – not low, however, but uniform, so that the cost of functions relative to each other is roughly the same, traced or untraced.

Sample output from the ctl file is shown in Figure 7.

The next section discusses the internals of the trace device, and then describe the tools we have developed for visualizing control flow of the kernel.

3.1 Trace device internals

Plan 9 devices follow a certain structure, consisting of a set of static functions and a structure which is initialized with those functions. This structure becomes

Command	parameters	description
trace	<start-address> <end-address> new <name>	Creates a new trace. trace <name>. The trace is set up but not enabled. Large batches of traces can be set up and enabled later.
trace	<name> remove	removes the trace named <name>
trace	<name> on	enables the trace named <name>
trace	<name> off	disables the trace named <name>
size	<size log 2>	resizes the kernel-based trace buffer to 2^{size} records
query	<address>	determines if the given address is traced. Useful for testing.
testtracein	<addr> <arg1> <arg2> <arg3> <arg4>	simulates entry into traced code. Useful for testing both the device and programs that process the data.
watch	<pid>	enables tracing on a process id (PID)-specific basis and further enables tracing on that one PID only.
start		enables tracing globally (the flag is tested in _profin/_profout assembly code)
stop		disables tracing globally (the flag is tested in _profin/_profout assembly code)

Table 1: Trace device commands

part of a linked list of device structures. None of the device functions are visible outside the device structure.

The trace device differs from most Plan 9 devices in that it additionally exports two *architecture-independent* C global functions, which are called from *architecture-dependent* assembly linkage functions. The architecture-independent, global device functions are *tracein* and *traceout:*

- tracein(void* pc, uintptr a1, uintptr a2, uintptr a3, uintptr a4)
- traceout(void* pc, uintptr retval)

. The uage of trace and traceout is outline in Figure 10.Their function is to conditionally add a trace record to a record buffer. A single record element is defined as follows:

```
struct Tracelog {
    u64int ticks;
    int info;
    uintptr pc;
    uintptr dat[5];
};
```

A u64int is a 64-bit unsigned integer. The uintptr type is defined (in Plan 9) to be the basic machine int type that is large enough to hold a pointer – in other words, a type that can represent a program counter, i.e. a pointer, and an integer. This type is hence different on different architecures, both in element size, structure size, and endian-ness.

This structure represents the internal tracelog format and, unlike the Linux devices, is not exported from the kernel to the user as such. Exporting this binary data directly would make the device interface architecture-dependent. Plan 9 is a distributed operating system designed for heterogeneity, and data such as this is always delivered to user mode in an architecture-neutral format – both endian and word-size independent. Architecture-neutral data formats allows one host to *import* another host's device files and operate on them directly. The format exported to user mode is textual, as described earlier.

As mentioned, the Tracelog records are accumulated into a trace buffer, which defaults to holding 8192 trace entries. The trace buffer is always a power of 2 in size, to ease the computation of FIFO pointers. The trace device keeps a write (pw) and read (pr) pointer, with low order bits being used as an index into the trace buffer. If an overrun occurs, old records are overwritten with new ones. The values of the pr and pw pointers are available from the ctl file, so overrun is easily detected.

The tracein function:

- checks to see if the PC is traced
- checks to see if we are tracing specific processes and, if so, if this process is traced

```
struct Trace {
        struct Trace *next;
        void *func; /* function being traced */
        void *start; /* start address of trace range */
        void *end; /(* end address of trace ranged */
        int enabled;
        char name[16];
};
```

Figure 8: The trace structure

- allocates a trace record (and returns immediately if that fails). Trace record allocation is done by an atomic increment of pw, and hence is cheap and SMP-safe.
- Fills the record in with the information

The traceout function is essentially identical, save that the information it logs is the return value from the function.

Determining whether a Program Counter (PC) is traced

For determining whether a PC is traced, we use a direct-mapped array of pointers to trace structures, called the *trace map*. The trace device trace structure is shown in FIgure 8.

On the first open of the device, the device allocates a trace map that has as many elements as bytes of code in the kernel. As trace structures are created via the trace command, we set pointers to the trace structure in the indices of the array representing traced areas of the kernel. For example, a trace structure for addresses 0x80100030 to 0x80100040 would result in indices 0x30 to 0x40 being populated with a pointer to that Trace structure. To simplify the book-keeping, we explicitly disallow overlapping trace ranges – including duplicate ranges. To check whether a PC is traced, we need merely compute the offset of the PC from the start of the kernel, and, using that as an index into the array, see if that array element has a pointer to a Trace struct. This approach trades storage for time and code complexity; given the the Plan 9 code space is less than one megabyte, it is a good trade.

Determining whether a process is traced

The device maintains a list of traced PIDs. If the array is not empty, signified by a 'traced PID count' being > 0, then it looks the PID up in the array to see if it is traced. In any event, a process that has opened the trace file will not be traced.

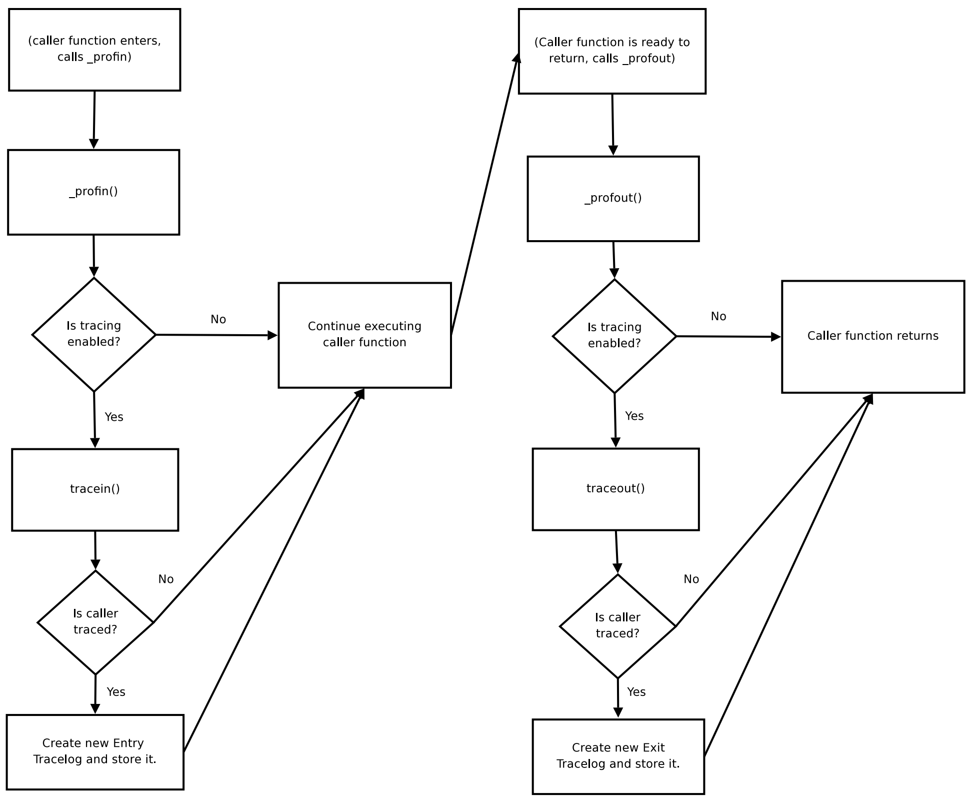
(caller function enters, calls _profin)
_profin()
Is tracing enabled?
No
Continue executing caller function
Yes
tracein()
Is caller traced?
No
Yes
Create new Entry Tracelog and store it.
(Caller function is ready to return, calls _profout)
_profout()
Is tracing enabled?
No
Caller function returns
Yes
traceout()
Is caller traced?
No
Yes
Create new Exit Tracelog and store it.

Figure 9: Flow of trace functions

Returning the data

Processes wishing to read trace records open the trace file and read from it, via the read system call, which in turn maps into a read function in the trace device.. The trace device read function, given a read request, formats as many records as will fit into the read output buffer, and returns.

3.1.1 The difficulties of simplicity

As we have noted, the trace device interface and the device itself are small: the C code is 800 lines, or roughly 1/3 the size of the Linux kprobes code; the assembly is less than 50 lines. The device described is the result of several iterations, not on just Plan 9, but on Linux. We started with a complex device that rewrote kernel code, based on our earlier work with DKM. The complexity of that code, comparable in scope to kprobes or djprobes, led us to look for a better and simpler (albeit less capable) design.

One of our goals is to make the trace data easily accessible to users. Linux has done an impressive job in this area with tools such as SystemTap. But, in the end, the Linux tools are very complex systems that are put in place to control other very complex systems. Using the raw interface of kprobes is a daunting task, requiring a lot of knowledge of the users. SystemTap eases the pain, but at the cost of comprehension when things do not go as planned.

In contrast, tracedev follows the Plan 9 path of a simple, regular device interface that can be directly used – even from the shell or command line. We regularly control tracedev by echoing commands into the ctl file and using cat to read the data file. The power of this interface is hard to overstate. Any tool that can process textual data can pull data from the trace device and process it.A non-expert can easily use tracedev to monitor Plan 9.

4 Usage examples

In this section we discuss the processing pipeline which produced the graph in Figure 1; a quick analysis of IO sizes for an interactive Plan 9 session; and an answer to the question: "What addresses are used when a process communicates with the kernel? Can we cache translations to speed up system calls?".

4.1 Visualizing trace device output

Once we had the data, we needed a way to analyse the information. After working with the data for a while, we realized that the output as shown in Figure 1 would be very useful. No graphiing tool available to us in Plan 9 or Linux was able to create that output. In the end, we determined that gnuplot was the most appropriate tool, but even then the data required significant processing to get it into the proper form.

We wrote a suite of scripts usng *rc*, the plan 9 shell; *acid,* the Plan 9 debugger; *awk*, and *sed* to generate data appropriate for plotting with gnuplot. The

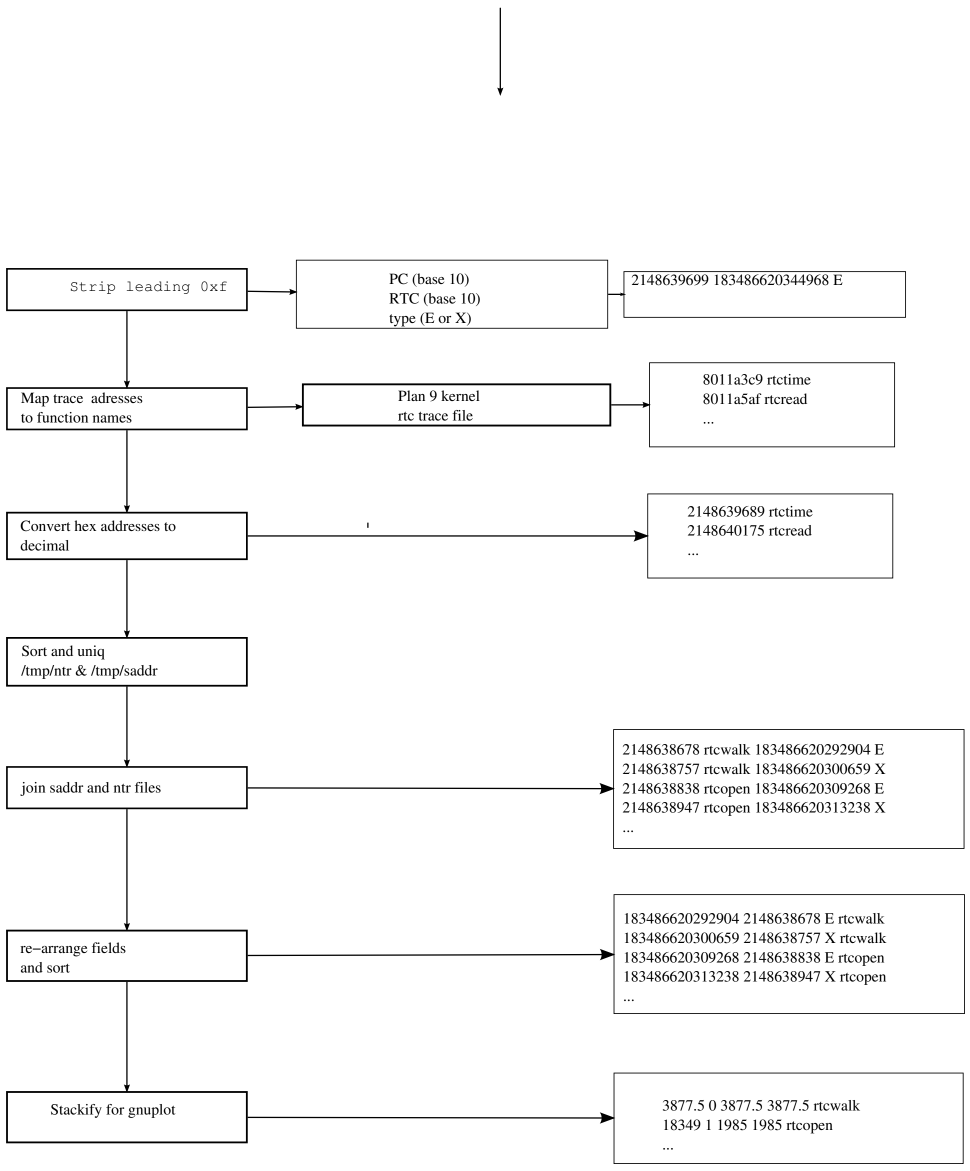
Strip leading 0xf
PC (base 10)
RTC (base 10)
type (E or X)
2148639699 183486620344968 E
Map trace adresses
to function names
Plan 9 kernel
rtc trace file
8011a3c9 rtctime
8011a5af rtcread
...
Convert hex addresses to
decimal
2148639689 rtctime
2148640175 rtcread
...
Sort and uniq
/tmp/ntr & /tmp/saddr
join saddr and ntr files
2148638678 rtcwalk 183486620292904 E
2148638757 rtcwalk 183486620300659 X
2148638838 rtcopen 183486620309268 E
2148638947 rtcopen 183486620313238 X
...
re−arrange fields
and sort
183486620292904 2148638678 E rtcwalk
183486620300659 2148638757 X rtcwalk
183486620309268 2148638838 E rtcopen
183486620313238 2148638947 X rtcopen
...
Stackify for gnuplot
3877.5 0 3877.5 3877.5 rtcwalk
18349 1 1985 1985 rtcopen
...

Figure 10: Processing pipeline

```
echo trace 17a099 17a0a2 new pr > /dev/tracectl
echo trace 17a2cf 17a2d8 new pw > /dev/tracectl
echo trace pr on > /dev/tracectl
echo trace pw on > /dev/tracectl
```

Figure 11: Command for tracing pread and pwrite

createplot script has the ability to filter out functions which ran for less than a specified number of clock cycles, which is useful for reducing the amount of noise in a plot. To generate a plot from the data collected earlier, discarding functions which completed in less than 4000 cycles, we just ran:

```
plots/createplot /amd64/9k8pf 4000 ./trace > plotme
```

and fed the input into gnuplot.

4.2 Example 2: What are typical IO sizes?

Plan 9 I/O is slower than we would like. We might want to speed it up, but first, we ought to know what sizes of I/O operations occur. All I/O goes through the pread and pwrite system calls. To get a quick idea of what might be going on, we decided to monitor only the return values of these calls.

We set up tracing as shown in Figure 11.

These commands are in a script. In one window, we run a small program which, in the inner loop, reads the trace data and builds a histogram. We show this inner loop in Figure 12. While this program is running, we started a window-based editor and viewed programs and copied files, which roughly corresponds to an 'interactive workload'.

When the user sends the program a signal (or hits the Plan 9 equivalent of ˆC) then the program dumps the histogram and exits. The result (with a little help from Octave) is shown below. Simply put, almost 90% I/O operations are less than 1Kbyte; 1/3 are under 8 bytes. An I/O enhancement strategy designed around these figures could greatly improve performance.

4.3 Example 3: What addresses are used when a process communicates with the kernel? Can we cache translations to speed up system calls?

We consider the case of processes communicating with the kernel. In earlier work, which showed us the need for the trace device, we determined that getting user data into and out of the kernel was a costly process. A large fraction of this time is mapping user level pointers into the kernel address space so that copying can be done. We speculated that If we can cache frequently used mappings, we might improve performance, particularly for read and write. For this simple test, we measure `tar cf /dev/null /sys/src`.

```
while (1) {
    char *cp;
    char *fields[16];
    cp = readline();
    if (! cp)
          continue;
    getfields(cp, fields, 16, 1, " ");
    size = strtol(fields[4], 0, 16);
    if (size < 0)
           bad++;
    else if (size > 16384)
           histo[16384]++;
    else
           histo[size]++;
}
```

Figure 12: Program fragment for creating a histogram from the trace device data file. This program is monitoring all reads and writes from other programs on the system.

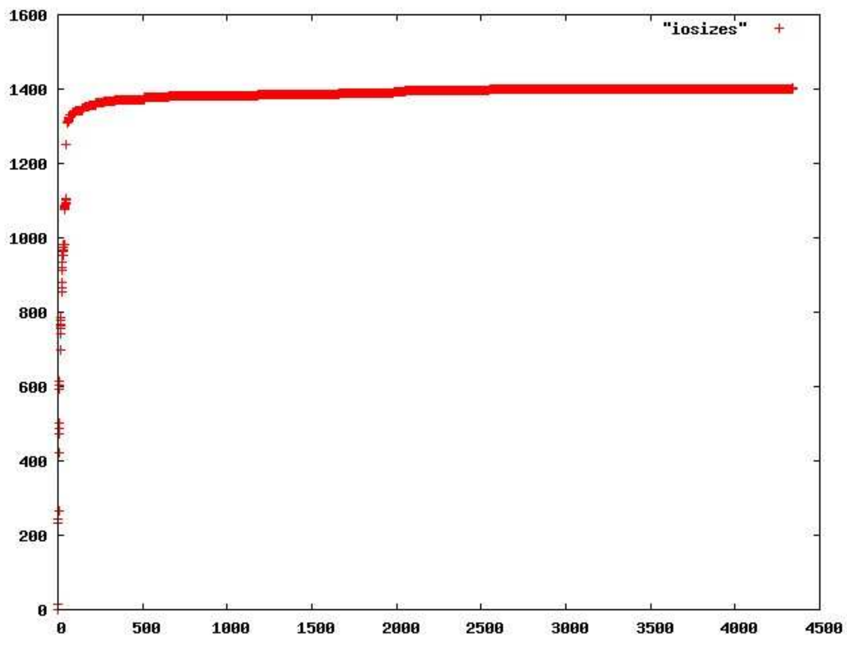

Figure 13: I/O sizes for an interactive Plan 9 session on a network terminal. The X axis is I/O size and the Y axis is a cumulative count of the number of I/Os. Of the total of 1400 I/Os, 1380 of them were less than 500 bytes.

Processes communicating with the kernel pass a user-mode virtual address to several system calls. It would be a bit of work to set up the 20 or so triggers, but fortunately there is a function in the kernel, *okaddr*, which is called to validate user addresses. Hence we can watch the parameters to the okaddr function. The success of a caching strategy is critically dependent on the number of different addresses used – if the number is small enough (e.g. 32), we can easily cache address mappings; if the number is too large (e.g. 16384) then it is unlikely that caching is practical.

For this test, our program records the PID and the address passed to *okaddr*. We only monitor function entry. The program further converts the address to a page address (i.e. divide by 4096) since that is the granularity of the value that is checked by the kernel and also the granularity of the mapping that would be cached. The results are shown in Table 2. The very large numbers are stack addresses.

The results show that we could cache as few as 32 page address translations for a process and eliminate much of the cost of both checking a virtual address and converting it to a physical address for kernel I/O.

5 Performance

The performance impact of any tracing system is always of concern. To test performance, we set up a commnand to copy data from /dev/zero to /dev/null, on megabyte at a time, in a kernel compiled without profiling, one compiled with profiling but with tracing turned off, and then again on the profiling kernel with tracing enabled. A kernel was also compiled with profiling enabled and the assembly-level _profin and _profout functions executing RET (return) immediately upon entry. Table 3 shows the results.

There is a very clear difference in performance between each test. Simply using a kernel compiled with tracing resulted in a 28% time increase over the non-profiling kernel. Going from a profiling kernel with tracing disabled to a profiling kernel tracing a section of memory gave a 74% increase in real time. Interestingly, having _profin and _profout return immediately gives about the same 28% hit as using a profiling kernel with tracing disabled; this is likely due to the pipeline being cleared by the CALL instruction. We may need to further modify the linker to either inline the functions or find some other way to make profiling-disabled functions more efficient.

However, when the intended use of the the trace device is taken into account, i.e. comparative measures of internal kernel function performance, these performance hits are not particularly problematic. Tracing is useful for gaining an idea of which functions take longer to execute. Since tracing creates an equal penalty for all traced functions, the ratios of execution times will still remain the same. As the plot in Figure 1 indicates, useful information can be gleaned without even knowing the time scales involved – it is sufficient to simply look at the graphs and see the different lengths of function execution.

The implementation we chose focuses on portability, and the avoidance of

PID	page address	count
102	4294967294	6
150	4294967294	6
176	15	63
176	16	4087
176	17	556
176	18	484
176	19	188
176	20	254
176	21	445
176	22	431
176	23	566
176	24	51
176	27	43
176	28	264
176	29	201
176	30	104
176	31	121
176	32	166
176	33	68
176	36	5
176	37	61
176	38	131
176	39	6
176	40	3
176	4294967294	14034
178	4294967292	3
178	4294967294	3
193	4294967294	3
199	33	3
199	34	2
199	4294967292	8
199	4294967294	6
51	1	3
51	10	3
51	18	1
51	19	2
51	38	44
51	4294967294	2

Table 2: kernel addresses used for tar, file server, and other server processes for tar pipeline

Profiling level	User	System	Real	% penalty
None	2.94	12.07	15.76	0
Included but disabled	3.87	16.34	20.24	28
Tracing I/O system calls	5.23	30.08	35.32	74
_profin and _profout returning immediately	4.13	16.57	20.71	28

Table 3: Performance for various levels of tracing

self-modifying code (which is what Kprobes, DJprobes, and Dtrace really are), while incurring a penalty in performance for untraced functions. It would be possible to write a simpler _profin function which pushes the contents of the BP register to the stack and calls tracein. The tracein prototype would then be

```
tracein(uintptr arg1, void* pc, uintptr ig-
nore, uintptr a2, uintptr a 3, uintptr a4);
```

note the `uintptr ignore`, which is a side affect of the way arguments are stored in the AMD64 architecture (the first argument is put into the BP register, but a slot is still left for it on the stack). Such a scheme would ultimately save 14 MOVQ, PUSHQ, and POPQ instructions. The trick is to make an architecture-independent tracein function that can be called the same way for all architectures. We are not sure this trick is possible.

6 Performance optimization

The current design uses the Plan 9 linker to insert an always-executed call to _profin and _profout at entry and exit points. As noted, this adds a fixed cost of almost 28% to common kernel operations. Hence, we can not ship a kernel with tracing always enabled, as Sun does. The question must be asked: could we change how we enable tracing? It turns out we can, if we are willing to consider using a rewrite-based approach. As it happens, we can make this approach efficient, SMP-safe, and not require additional code buffers for saving and restoring function code. We can completely eliminate the 'invalid kernel state' problem that the other code rewrite systems have.

The linker currently emits the following code, for every function:

```
CALL _profin(SB)
```

We can modify the linker to emit a slightly different sequence:

```
BR .+7
CALL _profin(SB)
```

The result would be that calls to profin would never happen. In order to enable the call to profin, one would rewrite the branch (either before the kernel is booted, or from the trace device) as follows:

```
    BR .+2
    CALL _profin(SB)
```

The function return case is easy: instead of

```
    CALL _profin(SB)
    RET
```

We have the linker emit:

```
    RET
    CALL _profin(SB)
    RET
```

To trace-enable a return, we simple change the RET to a NOP. The cost for the non-trace-enabled return is zero.

The only potential concern is the cost of the added branch on function entry: every function will have an added BR .+7 as the first instruction. This seems like it ought to slow things down. As it turns out the penalty is not nearly as bad as we might expect. Initial benchmarks showed encouraging results. We modified the Plan 9 loader to emit this sequence by default for profiled kernels, and a number of use-based benchmarks showed a 3 percent overhead, and as low as 1.5 percent on an Opteron. In fact we are using this kernel almost continuously now as the performance impact is really not noticeable.

In this code rewrite system, unlike the others mentioned, *the kernel code is never in an invalid state*: it transitions from one valid state to another. There is no need for an external code buffer, multiprocessor synchronization as the probes are installed and removed, or all the other complex overhead of the other rewrite systems. This design represents a substantial improvement over other trace devices, combining the best attributes of most of them: minimal overhead when not enabled; no invalid kernel state; and low cost for inserting a probe

7 Conclusions and future work.

Devtrace is a kernel trace device for Plan 9. It follows the Plan 9 model of providing a simple, textual control interface that requires no C code or even programming on the users part. It differs from other efforts in that it does not use complexity to hide complexity; rather, it is a very simple device. We showed two possible implementations. The first requires no self-modifying code as many other trace devices do. It does extract a high performance penalty, however. The second implementation extracts a measured penalty of 1.5% (AMD Opteron in 64-bit mode) or 3% (Intel Xeon in 32-bit mode). The second does require self-modifying code, but not the unsafe self-modifying code used in, e.g., Kprobes or DJProbes: the kernel code never makes a transition from valid to invalid to valid, but rather only makes a transition between two valid states.

We showed a number of uses of the trace device, including overhead measurement, as well as I/O size measurement. Finally, we showed that the kernel could implement a cache for virtual addressed that would be effective with as few as 32 entries. The I/O size measurements and the virtual address measurement point to ways to greatly improve I/O performance while maintaing the Plan 9 I/O model.

This work has relevance to other operating systems as well. We could modify other compilers, such as gcc, to emit the performance-optimized trace calls shown above. It would be easy to have gcc generate the instrumentation for a Linux kernel. In some ways the gcc work would be easier; we would not need to write the assembly code interface, but could, rather, have gcc generate it.

Future work includes improving the display of results, as well as providing a more automated interface.

Appendix A: Alternate Implementations

There are a number of decisions we made in this design that, in retrospect, we could take differently. There was also an earlier rewrite-based design that we stopped developing.

7.1 Earlier implementation: full code rewrite

It is worth mentioning an earlier implementation of devtrace. The interface was unchanged, save that we added PID selection to the later device. But the implementation was somewhat different than the approaches taken above. We show the flow in Figure 14. We rewrote the code for the entry point. To gain control at function exit, when the code buffer code was executed, we saved a copy of the return PC in the code buffer and modified the return PC on the stack. The code buffer thus has a dual role: saving state and running the trace code.

Once the code buffer was entered, it called the trace function, restored the function code, modified the return PC on the stack, and then jumped back to the function. On function exit, code buffer code was again executed, to re-insert the jump, call trace code, restore the PC on the stack and, finally, exit. This methodology was reasonably solid on a single processor and imposed no penalty on performance for non-traced functions. On inspection, it obviously has a lot of opportunity for failure on an SMP. Additionally, it will not work for re-entrant functions. In fact there are so many opporunities for failure in this design that it represents more of a curiousity than a working solution. Finally, at almost 100 instructions on the Power PC, the overhead for a traced function was quite high.

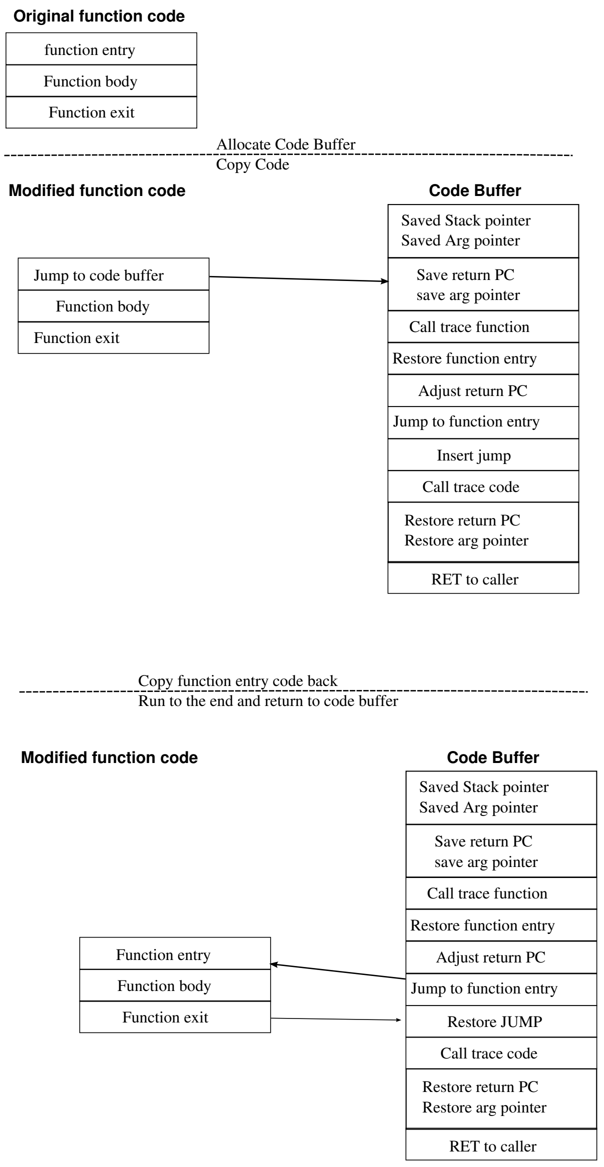

Figure 14: Flow for the rewriting version of devtrace

7.2 Data format

Our data format is text, not binary. We might be called to task on this question: it is taken as given that binary data is always more efficient to transfer than text, and our fixed format is maximally inefficient. Recall that our trace struct is this:

```
struct Tracelog {
        u64int ticks;
        int info;
        uintptr pc;
        uintptr dat[5];
};
```

On an Opteron this struct is 64 bytes. Our inefficient text format is double this, at 128 bytes. Realistically, however, we can shrink the text-based output quite a bit if we abandon fixed-format records. Recall the the E format is:

E PC clock PID arg1 arg2 arg3 arg4

The PC can be printed as an offset from kernel code base, and hence limited to 5 bytes. We will assume the clock remains 8 bytes. The PID can certainly be less than 16 bytes, and can be as small as 4 bytes (PIDs in the range 0-9999). The args are generally small, and could again be less than 4-5 bytes each. An E record can, hence, be as small as 35 bytes – half the size of the binary!

The X record can be even shorter, as there is one return value to print, not four args: X records could be under 20 bytes.

If we move to a non-fixed format record, we can conceivably *increase* the bandwidth and efficiency of the data path as compared to binary. The question of the relative convenience of the two formats remains an open question.

References

[1] Andrew McRae, *Hardware profiling of kernels, or: How to look under the hood while the engine is running.*

[2] Satoshi Oshima, *Djprobes status.*

[3] Matthew J. Sottile and Ronald G. Minnich, *Supermon: A high-speed cluster monitoring system*, CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing (Washington, DC, USA), IEEE Computer Society, 2002, p. 39.