Measuring kernel throughput on Blue Gene/P with the Plan 9 research operating system [*]

Ronald G. Minnich and John Floren
Sandia National Labs
Aki Nyrhinen
University of Helsinki, Department of Computer Science

October 12, 2009

Abstract

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. Part of our research is to answer the following question: can a full-featured operating system like Plan 9 equal the performance of a lightweight kernel such as IBM's Compute Node Kernel (CNK)? We expected the CNK to provide better performance than Plan 9 in several areas, e.g., CNK supports 1 Mbyte pages and Plan 9 only uses 4096 byte pages. This page-size difference will, in turn, result in better performance for applications which have highly non-local memory reference patterns.

It is critical that we be able to finely measure kernel performance. While such systems are taken for granted in the Unix world, to date, Plan 9 has only supported a profiling interface, not a tracing interface. Profiling interfaces can provide gross information about "where the time is spent" across all processes using the machine. In essence, a profiling interface removes all time and call tree information, providing only a summary. A tracing interface is able, for a given process, to show exactly what functions called each other, and when.

We have created a Plan 9 trace device, devtrace, which 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

[†]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. SAND-2008-4108-P. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

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.

We have used this device to quantify the advantage of our 1M page implementation in Plan 9. We are further using it to optimize the I/O path from process to network.

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 system can easily measure their system's performance.

1 Introduction

This project started out as a simple question: where is the time going in Plan 9, and why? 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.

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 script processing pipeline. The trace shows the kernel functions called by an 'echo' command. The X axis is the time in processor ticks; the Y axis is each kernel function, by name. For readability, we have filtered out all kernel functions that take less than 50,000 ticks. The red line marks the time a function is entered until the time it exits. The calling hierarchy can be determined by seeing what lines overlap other lines. For example, we can see that the syspwrite function calls pwrite, as they cover the same part of the X axis, and syspwrite is wider. Functions called multiple times would appear multiple times on this graph (they have been filtered out for readability).

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

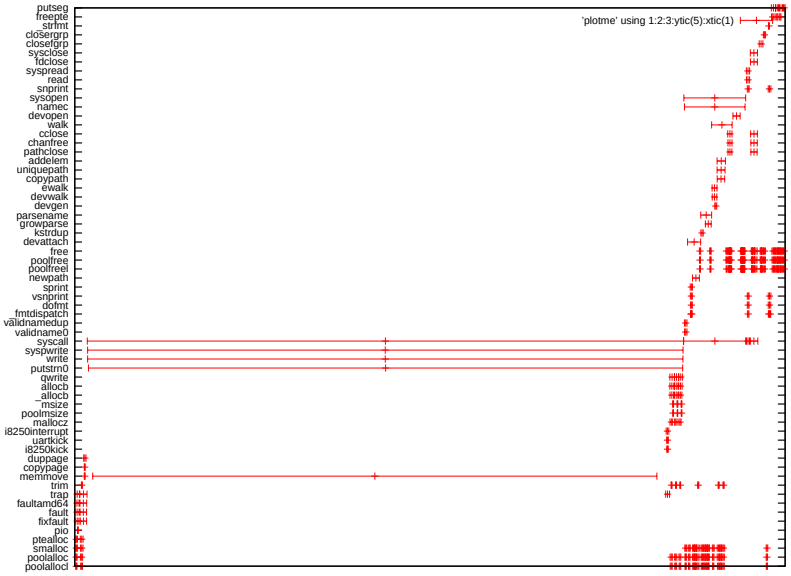

Figure 1: A sample trace output with tracedev. The X axis is in units of processor ticks.

Stride	CNK	Plan9	Plan 9 (1MB page)
1	3.7539700e-01	3.8000000e-01	3.7000000e-01
101	3.7721600e-01	1.6400000e+00	3.8000000e-01
201	3.7926300e-01	3.4000000e+00	3.8000000e-01
301	3.8119100e-01	4.9100000e+00	3.8000000e-01
401	3.8263600e-01	6.3300000e+00	3.8000000e-01
501	3.8444400e-01	7.7600000e+00	3.9000000e-01
601	3.8661300e-01	7.7700000e+00	3.9000000e-01
701	3.8841900e-01	7.7600000e+00	3.9000000e-01
801	3.8986500e-01	7.7600000e+00	3.9000000e-01
901	3.9227500e-01	7.7700000e+00	3.9000000e-01
1001	3.9444300e-01	1.1750000e+01	4.0000000e-01

Figure 2: Comparative performance of: CNK; Plan 9 with 4K pages; Plan 9 with 1 MB pages. Even with tracing on, Plan 9 is sometimes faster.

device and control its actions; the trace file is used to read trace records.

Example usage

We wanted to measure the performance of SAXPY on the Blue Gene/P (BG/P) system at Argonne, on both the CNK and Plan 9 kernels. We used the strid3 benchmark from Lawrence Livermore National Labs. Strid3 does multiple SAXPY operations, with different strides each time. The benchmark "is designed to severely stress the memory subsystem on a node ...utilize[s] combinations of loops of scalar and vector operations and measure[s] the MFLOP rate delivered as a function of the memory access patterns and length of vector utilized". Strid3 was of particular interest as it can expose problems with both TLB and cache because, at larger strides, it can invalidate both the TLB entry and a cacheline *for each memory reference*. Relative performance is shown in Figure 2; we use the time trace output from strid3.

The initial results were not encouraging: as the strides grew larger, runs under Plan 9 showed a factor of 30 penalty over CNK. We suspected that the problem was our use of 4096-byte TLBs as opposed to the CNK's use of 1 Mbyte TLBs. To get some idea of the number of faults for strid3, we counted them.

To count the faults we had to set up a trace for the *faultpower* function in the kernel, which is always called on a TLB miss. Using nm we obtain the address range of faultpower and execute the following command:

```
cpu% echo trace 0xf00049f4 0xf0004b38 new faults  > /dev/tracectl
```

This command creates a new trace, called *faults*, which covers the range selected by the two numbers.

Traces are not enabled by default. We can create any number of traces, and have them enabled and disabled independently. To enable a trace:

```
echo trace faults on > /dev/tracectl
```

We also have to actually enable the trace device itself, with the *start* command:

```
echo start > /dev/tracectl
```

This setup is very similar to a logic analyzer setup. We create the triggers, enable some of them, then start the logic analyzer. After the strid3 run, we can see how many page faults occurred:

```
cat /dev/tracectl
```

which shows:

```
logsize 16
trace f0004944 f0004a88 new fault
#trace f0004944 traced? f16a33d0
trace fault on
#tracehits 863700, in queue 863700
#tracelog f1909d90
#traceactive 0
#slothits 293803185
#traceinhits 131995046
watch 0
```

This output, as for most plan 9 devices, consists of a valid set of commands and comments (comments start with a #). The output can be used as input to the device. It is very easy to save the output and, later, rerun the trace.

We discuss the commands later in this paper; for now, the main item of interest is 'traceinhits', i.e. the number of times we had a hit on an active trace. It shows that strid3 took 132 million TLB miss faults. It's certainly reasonable to guess that we might improve performance by reducing the number of faults.

To do so, we need to implement one Mbyte TLB entries, a.k.a. "huge pages". The way in which we implement this change is grist for another paper, but we can say that the careful addition of 12 lines of code is sufficient to add them. We perform the run again, with much better results.

We can get immediate confirmation of this improvement from the trace output:

```
logsize 13
trace f00049f4 f0004b38 new fault
#trace f00049f4 traced? f169d4f0
trace fault on
#tracehits 9918, in queue 9917
#tracelog f1894440
#traceactive 0
#slothits 18639
#traceinhits 9535
watch 0
```

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

Figure 3: Data output from /dev/trace

We can see a 10,000-fold reduction in the number of calls to faultpower. The dramatic improvement in strid3 can be traced to this reduction. On an application-by-application basis, we are using devtrace to wring out the performance issues in Plan 9 so that this full-featured kernel can equal the performance of a lightweight kernel.

For this case, there was no need to look at the actual trace records themselves. We will describe the format here in brief. The trace records are also text, which facilitates cross-platform processing of the data: this interface is inherently heterogeneous.

To read the trace records, we type:

```
cat /dev/trace
```

and can see the records shown in Figure 3.

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. Finally, we close with a description of future work.

2 Related work

There have been many implementations of kernel tracing facilities over the years. 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

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

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.

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 3. 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 overwritten code in it, and installs a jump to the code buffer in place of the overwritten code. The buffer contains overwritten code, a call to the trace support code, and code that resumes the function. 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(2), 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 requires interrupt handling.

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.

As might be expected, supporting self-modifying code is a very complicated process, when all the possible execution paths are taken into account. Multiprocessor machines further 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 (4). Further 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; is it ever safe to remove a probe?

2.1.1 dtrace

This section would be incomplete without a reference to Sun's dtrace(1), 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.

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
- 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. 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 – in fact, it shares 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.

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 (5). 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 (5). 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.

Dtrace, and our Plan 9 trace device (devtrace) opt for a fixed-format, fixed-size data format, for reasons of processing complexity and overhead. Devtrace also fixes 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.

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. We can trace 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. 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 and _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.

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 returns trace records as text.

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.

3.0.1 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. 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. Our work paid off: it took two hours to port the trace device from the AMD 64 kernel to the BG/P kernel.

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

```
tcfd = open("/dev/tracectl", ORDWR);  /* open the trace device */
tree = open("/dev/vc0net", ORDWR);    /* open the tree network */
m = smprint("watch %d", getpid());    /* command to watch ourselves */
write(tcfd, m, strlen(m));            /* write watch self command */
write(tcfd, "start", 5);              /* start tracing */
write(tree, "hi", 2);                 /* do tree I/O */
write(tcfd, "stop", 4);               /* stop tracing */
```

Figure 4: The test program

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 Use cases

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

4.1 Example 2: how long does it take to get from a process to an I/O device?

We wish to know how long it takes a process to write to an I/O device, as we need to reduce this time to be competitive with MPI. The test program is shown in Figure 4. The simplicity should be apparent: open a control file and device; write commands to watch the process and then start the trace run; write to the tree; stop tracing.

The results are shown below. The write is currently not nearly fast enough; the writes hit the wire at the beginning of txstart, which on close inspection is almost 90,000 ticks in. That said, we do hit the treewrite function at only a few thousand ticks in; the opportunities for optimization are pretty obvious.

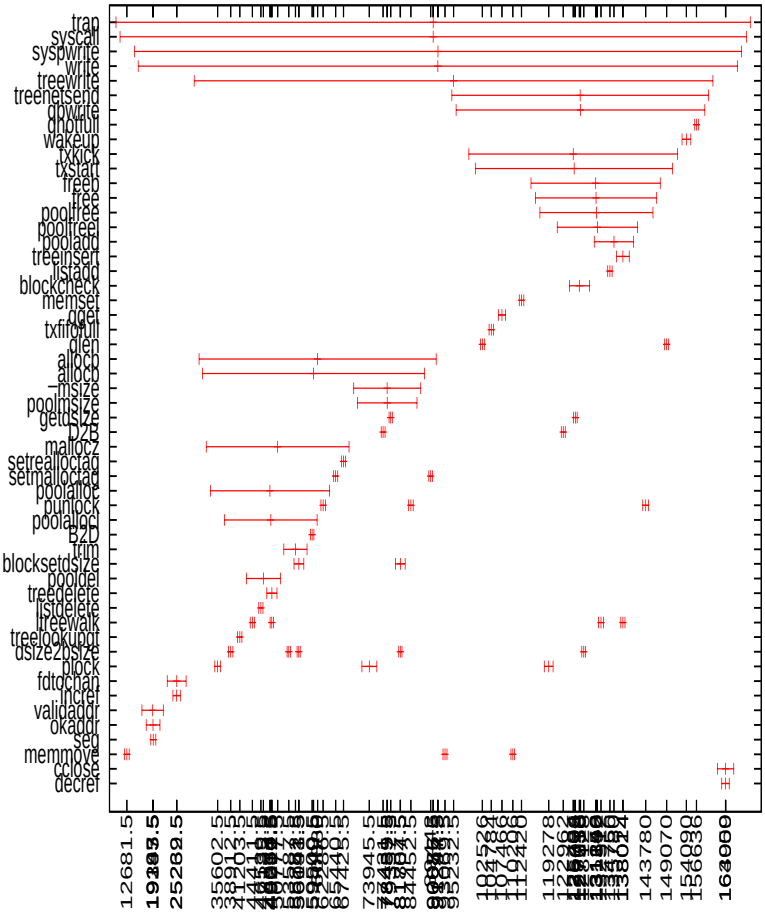

Figure 5: Results of watching tree network write

```
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 6: Command for tracing pread and pwrite

4.2 Example 3: What are typical IO sizes?

We wished to know what size I/O operations programs initiate. 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 6.

The result is shown in Figure 7. Almost 90% of 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 4: 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`.

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 page address passed to *okaddr*. We only monitor function entry.

The results shown in Table 2 indicate that we can 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.

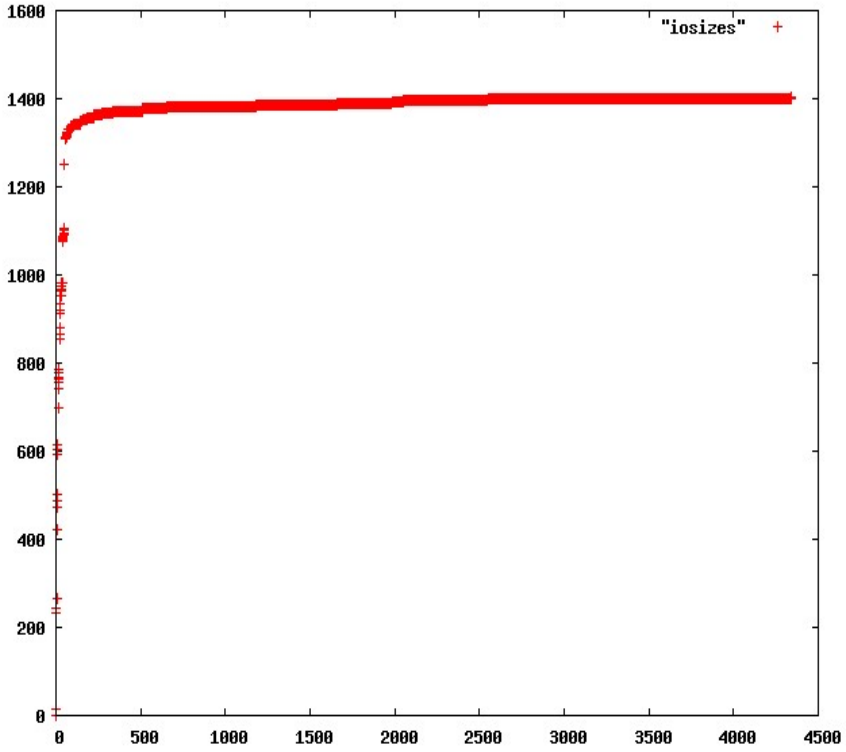

Figure 7: I/O sizes for an interactive Plan 9 session on a network terminal. The X axis is I/O size and the Y axis is a cumulative count of the number of I/Os. Of the total of 1400 I/Os, 1380 of them were less than 500 bytes.

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

5 Performance

The performance impact of any tracing system is always of concern. To test performance, we set up a comnmand 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.

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, since the code change is only one byte and by definition can not span cache lines.

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 _profout(SB)
RET
```

We have the linker emit:

```
RET
CALL _profout(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 our implementation of a code rewrite system, unlike the others described above (dtrace, kprobes, djprobes), *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.

We have shown a kernel device for measuring Plan 9 kernel overhead on the IBM BG/P supercomputer. The device, devtrace, follows the Plan 9 model of providing a simple, textual control interface that requires no C code or even programming on the user's 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 very low penalty. 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 system call latency measurement, overhead measurement, and I/O size measurement. Finally, we showed that the kernel could implement a cache for virtual addresses 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.

References

[1] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal, *Dynamic instrumentation of production systems*, 2004, pp. 15–28 (English).

[2] R. Krishnakumar, *Kernel korner: kprobes-a kernel debugger*, Linux J. **2005** (2005), no. 133.

[3] Andrew McRae, *Hardware profiling of kernels, or: How to look under the hood while the engine is running*, (1993).

[4] Satoshi Oshima, *Djprobes status*.

[5] 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.