Using Currying and process-private system calls to break the one-microsecond system call barrier

Ronald G. Minnich[‡] and John Floren and Jim Mckie

January, 2009

Abstract

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. In contrast to 20 years of tradition in High Performance Computing (HPC), we require that programs access network interfaces via the kernel, rather than the more traditional (for HPC) "OS bypass".

In this paper we discuss our research in modifying Plan 9 to support sub-microsecond "bits to the wire" (BTW) performance. Rather than taking the traditional approach of radical optimization of the operating system at every level, we apply a mathematical technique known as Currying, or pre-evaluation of functions with constant parameters; and add a new capability to Plan 9, namely, process-private system calls. Currying provides a technique for creating new functions in the kernel; process-private system calls allow us to link those new functions to individual processes.

1 Introduction

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. Our research goals in this work are aimed at rethinking how HPC systems software is structured. One of our goals is to re-examine and, if possible, remove the use of "OS bypass" in HPC systems.

OS bypass is a software technique in which the application, not the operating sytem kernel, controls the network interface. The kernel driver is disabled, or, in some cases, removed; the functions of the driver are replaced by an application or library. All HPC systems in the "Top 50", and in fact most HPC systems in the Top 500, use OS bypass. As the name implies, the OS is completely bypassed; packets move only at the direction of the application. This mode of operation is a lot like the very earliest days of computers, where not a bit of I/O moved unless the application directly tickled a bit of hardware. It involves the application (or libraries) in the lowest possible level of hardware manipulation, and even requires application libraries to replicate much of the operating systems capabilities in networking, but the gains are seen as worth the cost.

One of the questions we wish to answer: is OS bypass still needed, or might it be an anachronism driven by outdated ideas about the cost of using the kernel for I/O? The answer depends on measurement. There is not much doubt about the kernel's ability to move data at the maximum rate the network will support; most of the questions have concerned the amount of time it takes to get a message from the application to the network hardware. So-called short message performance is crucial to many applications.

HPC network software performance is frequently characterized in terms of "bits to the wire" (BTW) and "ping-pong latency". Bits to The Wire is a measure of how long it takes, from the time an application initiates network I/O, for the bits to appear on

[*] Sandia National Laboratories

[†] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under contract DEAC0494AL85000. SAND-2009-5156C.

the physical wire. Ping-pong latency is time it take a program to send a very small packet (ideally, one bit) from one node to another, and get a response (usually also a bit). These numbers are important as they greatly impact the performance of collectives (such as a global sum), and collectives in turn can dominate application performance [2] [4] In an ideal world, ping-pong latency is four times the "bits to the wire" number. Some vendors claim to have hit the magical 1 microsecond ping-pong number, but a more typical number is 2-3 microseconds, with a measured BTW number of 700 nanoseconds. However, these numbers always require dedicated hosts, devices controlled by the application directly, no other network activity, and very tight polling loops. The HPC systems are turned into dedicated network benchmark devices.

A problem with "OS bypass" is that the HPC network becomes a single-user device. Because one application owns the network, that network becomes unusable to any other program. This exclusivity requires, in turn, that all HPC systems be provisioned with several networks, increasing cost and decreasing reliability. While the reduction in reliability it not obvious, one must consider that the two networks are not redundant; they are both needed for the application to run. A failure in either network aborts the application.

By providing the network to programs as a kernel device, rather than a set of raw registers, we are making HPC usable to more than just specialized programs. For instance, the global barrier on the Blue Gene systems is normally only available to programs that link in the (huge) Deep Computing Messaging Facility (DCMF) library or the MPI libraries[1], which in turn link in the DCMF. Shells are not MPI applications; it makes no sense whatsoever to turn the shell into an MPI application, as it has uses outside of MPI, such as starting MPI programs!

On Plan 9 we make the global barrier available as a kernel device, with a simple read/write interface, so it is even accessible to shell scripts. For example, to synchronize all our boot-time scripts, we can simply put `echo 1 > /dev/gib0barrier` in the script. The network hardware becomes accessible to any program that can open a file, not just specialized HPC programs.

Making network resources available as kernel-based files makes them more accessible to all programs. Seperating the implementation from the usage reduces the chance that simple application bugs will lock up the network. Interrupts, errors, resources conflicts, and sharing can be managed by the kernel. That is why it is there in the first place. The only reason to use OS bypass is the presumed cost of asking the kernel to perform network I/O.

One might think that the Plan 9 drivers, in order to equal the performance of OS bypass, need to impose a very low overhead – in fact, no overhead at all: how can a code path that goes through the kernel possibly equal an inlined write to a register?

While it is true that OS bypass has zero overhead in theory, it can have very high overhead in fact. Programs that use OS bypass always use a library; the library is usually threaded, with a full complement of locks (and lockiing bugs and race conditions); in the end, we have merely to offer lower overhead than a library.

There are security problems with OS bypass as well. To make OS bypass work, the kernel must provide interfaces that to some extent break the security model. On Blue Gene/P, for example, DMA engines are made available to programs that allow them to overwrite arbitrary parts of memory. On Linux HPC clusters, Infiniband and other I/O devices are mapped in with mmap, and users can activate DMAs that can overwrite parts of kernel memory. Indeed, in spite of the IOMMUs which are supposed to protect memory from badly behaved user programs, there have been recent BIOS bugs that allowed users of virtual network interfaces to roam freely over memory above the 4 gigabyte boundary. Mmap and direct network access are really a means to an end; the end is low latency bits to the wire, not direct user access. It is so long since the community has addressed the real issue that means have become confused with ends.

[1] MPI libraries are typically much larger than the Plan 9 kernel

2 Related work

The most common way to provide low latency device I/O to programs is to let the programs take over the device. This technique is most commonly used on graphics devices. Graphics devices are inherently single-user devices, with multiplexing provided by programs such as the X server. Network interfaces, by contrast, are usually designed with multiple users in mind. Direct access requires that the network be dedicated to one program. Multi-program access is simply impossible with standard networks.

Trying to achieve high performance while preserving multiuser access to a device has been achieved in only a few ways. In the HPC world, the most common is to virtualize the network device, such that a single network device appears to be 16 or 32 or more network devices. The device requires either a complex hardware design or a microprocessor running a real-time operating system, as in Infiniband interfaces: thus, the complex, microprocessor-based interfaces do bypass the main OS, but don't bypass the on-card OS. These devices are usually used in the context of virtual machines. Device virtualization requires hardware changes at every level of the system, including the addition of a so-called iommu [1].

An older idea is to dynamically generate code as it is needed. For example, the code to read a certain file can be generated on the fly, bypassing the layers of software stack. The most known implementaiton of this idea is found in Synthesis [3]. While the approach is intriguing, it has not proven to be practical, and the system itself was not widely used.

The remaining way to achieve higher performance is by rigorous optimization of the kernel. Programmers create hints to the compiler, in every source file, about the expected behaviour of a branch; locks are removed; the compiler flags are endlessly tweaked. In the end, this work results in slightly higher throughput, but the latency – "bits to the wire" – time changes little if at all. It is still too slow. Recent experiences shows that veyr high levels of optimization can introduce security holes, as was seen when a version of GCC optimized out all pointer comparisons to NULL.

Surprisingly, there appears to have been little other work in the area. The mainline users of operating systems do not care; they consider 1 millisecond BTW to be fine. Those who do care use OS bypass. Hence the current lack of innovation in the field: the problems are considered to be solved.

The status quo is unacceptable for a number of reasons. Virtualized device hardware increases costs at every level in the I/O path. Device virtualization adds a great deal of complexity, which results in bugs and security holes that are not easily found. The libraries which use these devices have taken on many of the attributes of an operating system, with threading, cache- and page-aligned resource allocation, and failure and interrupt management. Multiple applications using multiple virtual network interfaces end up doing the same work, with the same libraries, resulting in increased memory cost, higher power consumption, and a general waste of resources all around. In the end, the applications can not do as good a job as the kernel, as they are not running in priveleged mode. Applications and libraries do not have access to virtual to physical page mappings, for example, and as a result they can not optimize memory layout as the kernel code.

3 Our Approach

Our approach is a modification of the Synthesis approach. We do create curried functions with optimized I/O paths, but we do not generate code on the fly; curried functions are written ahead of time and compiled with the kernel, and only for some drivers, not all. The decision on whether to provide curried functions is determined by the driver writer.

At run time, if access to the curried function is requested by a program, the kernel pre-evaluates and pre-validates arguments and sets up the parameters for the driver-provided curried function. The curried function is made available to the user program as a private system call, i.e. the process structure for that one program is extended to hold the new system call number and parameters for the system call. Thus, instead of actually synthesizing code at runtime, we augment the process structure so as to connect individual user processes to curried functions which are

already written.

We have achieved sub-microsecond system call performance with these two changes. The impact of the changes on the kernel code is quite minor.

We will first digress into the nature of Curry functions, describe our changes to the kernel and, finally discuss the performance improvements we have seen.

3.1 Currying

The technique we are using is well known in mathematical circles, and is called currying [**?**]. We will illustrate it by an example.

Given a function of two variables, $f(x, y) = y/x$, one may create a new function, $g(x)$, if y is known, such that $g(x) = f(x, y)$. For example, if y is known to be 2, the function g might be $g(x) = f(x, 2)$.

We are interested in applying this idea to two key system calls: read and write. Each takes a file descriptor, a pointer, a length, and an offset. In the case of the Plan 9 kernel, we had used a kernel trace device and observed the behavior of programs. Most programs:

- Used less than 32 distinct pages when passing data to system calls
- Opened a few files and used them for the life of the program
- Did very small I/O operations

We also learned that the bulk of the time for basic device I/O with very small write sizes – the type of operation common to collective operations – was taken up in two functions: the one that validated an open file descriptor, and the one that validated an I/O address.

The application of currying was obvious: given a program which is calling a kernel function read or write function: $f(fd, address, size)$, with the same file descriptor and same address, we ought to be able to make a new function: $g(size) = f(fd, address, size)$, or even $g() = f(fd, address, size)$.

Tracing indicated that we could greatly reduce the overhead. Even on an 800 Mhz. Power PC, we could potentially get to 700 nanoseconds. This compares very favorably with the 125 ns it takes the hardware to actually perform the global barrier.

3.2 Connecting curry support to user processes

The integration of curried code into the kernel is a problem. Dynamic code generation looks more like a security hole than a solution.

Instead, we extended the kernel in a few key ways:

- extend the process structure to contain a private system call array, used for fastpath system calls
- extend the system call code to use the private system call array when it is passed an out-of-range system call number
- extend the driver to accept a fastpath command, with parameters, and to create the curried system call
- extend the driver to provide the curried function. The function takes no arguments, and uses prevalidated arguments from the private system call entry structure

4 Implementation of private system calls on Plan 9 BG/P

To test the potential speeds of using private system calls, a system was implemented to allow fast writes to the barrier network, specifically for global OR operations, which are provided through `/dev/gib0intr`. The barrier network is particularly attractive due to its extreme simplicity: the write for a global OR simply places a 1 in a specific part of memory. Thus, it was easy to implement an optimized path to the write on a per-process basis.

The modifications described here were made to a branch of the Plan 9 BG/P kernel. This kernel differed from the one being used by other Plan 9 BG/P developers only in that its portable `incref` and `decref` functions had been redefined to be architecture-specific, a simple change to allow faster

performance through processor-specific customizations. In other words, we are comparing our curried function support to an already-optimized kernel.

First, the data structure for holding fast system call data was defined in the `/sys/src/9k/port/portdat.h` file (from this point on, kernel files will be assumed to reside under `/sys/src/9k/`, thus `port/portdat.h`).

```
/* Our special fast system call struct */
struct Fastcall {
    /* The system call number */
    int scnum;
    /* A communications endpoint */
    Chan *c;
    /* The handler function */
    long (*fun)(Chan*, void*);
};
```

In the same file, the `proc` struct was modified to include the following declarations:

```
/* Array of private system calls */
Fastcall *fc;
/* # private system calls */
int fcount;
```

Next, `bgp/devgib.c` was modified to accept "fastwrite" as a command when written to `/dev/gib0ctl`. When the command is written, the kernel allocates a new `Fastcall` in the `fc` array, giving it a system call number and a channel pointing to the barrier network, then sets `(*fun)` to point to the `gibfastwrite` function and finally increments `fcount`. The return value of the `write` call is the system call number. Thus, to create a new private call, one might write:

```
int fd, scnum;
fd = open("/dev/gib0ctl", OWRITE);
scnum = write(fd, "fastwrite", 10);
close(fd);
docall(scnum);
```

Following the `write`, `scnum` contains a number for a private system call to write to the barrier network. From there, a simple assembly function (here called `docall`) may be used to perform the actual private system call:

```
TEXT docall(SB), 1, $0
    SYSCALL
    RETURN
```

When a system call interrupt is generated, the kernel typically checks if the system call number matches one of the standard calls; if there is a match, it calls the appropriate handler, otherwise it gives an error. However, the kernel now also checks the user process's `fc` array and calls the given `(*fun)` function call if a matching private call exists. In the case of the barrier device, it calls `gibfastwrite`, which does little more than write a 1 to a memory location. This avoids several layers of generic code and argument checking, allowing for a faster write.

5 Results

In order to test the private system call, a short C program was written to request a fast write for the barrier. It opens `/dev/gib0ctl` and writes `fastcall` to the file. Then, it calls a tiny assembly function with the resulting private system call number as an argument, which does a `SYSCALL` for that number and returns. The private system call is executed many times and timed to find an average cost per call. As a baseline, the traditional write call was also tested using a similar procedure.

Initial results are promising. With the traditional write path, it took approximately 3,000 cycles per write. Since the BG/P uses 850 MHz PowerPC processors, this means a normal write takes approximately 3.529 microseconds. However, when using the private system calls, it only takes around 620 cycles to do a write, or 0.729 microseconds. The overall speedup is 4.83, a 383The result is a potential ping-pong performance of slightly under 3 microseconds, which is competitive.

6 Conclusions and Future Work

We have managed the write side of the fastcall path. What remains is to improve the read side. The read side may include an interrupt.

7 Acknowledgements

This work is support by the DOE Office of Advanced Scientific Computing Research. IBM has provided support from 2006 to the present. 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.

References

[1] Muli Ben-Yehuda, Jimi Xenidis, and Michal Ostrowski. Price of safety: Evaluating iommu performance. June 2007.

[2] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[4] Y. Tanaka, K. Kubota, M. Sato, and S. Sekiguchi. A comparison of data-parallel collective communication performance and its application. *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, 0:137, 1997.