Towards Running Parallel Programs on the Bare Metal via Virtualization

James Swaine and Stephen P. Tarzia
Northwestern University

March 20, 2009

Abstract

Decades of parallel computing practice have proven that highly parallel code runs efficiently only when it has uninterrupted access to the hardware. We report on a project whose goal is to support compiling Data Parallel Haskell code into bootable disk images. Our toolchain layers the Data Parallel Haskell runtime system on top of the GeekOS operating system and Newlib C library. We explain how our toolchain, combined with a virtual machine monitor, can allow optimized low-level parallel code to be run in cooperation with a traditional OS.

1 Introduction

The proliferation of multicore architectures in the consumer computing market has created rapidly increasing demand for languages capable of using these architectures efficiently. Older parallel languages typically used in supercomputing environments (High Performance FORTRAN, MPI) are often too narrow in scope or too low-level to suit the needs of the general-purpose commodity software developer, creating the need for higher levels of abstraction [4].

High-level parallel programming language implementers often find themselves in the unique (and often unenviable) position of attempting to apply general concepts to intricately difficult problems in parallel computing, many of which vary greatly from one hardware implementation to another. These challenges are compounded in preemptive multitasking environments, in which an operating system scheduler may unexpectedly deprive a parallel operation of one or more processing nodes to allow other processes to execute. These issues are often overlooked in the performance evaluation of parallel programs and languages, and can often significantly contribute to degradation of performance in parallel applications [7]. It is our belief that parallel programs must be able to allowed to execute as close to "raw hardware" as possible to maximize performance, while coexisting with traditional operating systems popular in the consumer market (Windows, Linux, Mac OS X).

We contend that it is vitally important that any toolchain using this approach must provide the programmer with the convenience and expressive power inherent in modern, high-level parallel languages. These languages free the programmer from low-level concerns such as thread synchronization, deadlock avoidance, and memory layout - allowing a greater portion of development effort to be focused on the "task at hand".

Though a discussion of parallel language design is beyond the scope of this work, here we briefly mention several notable research efforts to provide such languages. The work perhaps most explicitly targeting this goal is the Manticore project, which adds parallel primitives to the Standard ML functional programming language. Somewhat unique to the project is the presentation of a unified API for "heterogeneous" parallel operations, encouraging use of different types of parallelism (coarse-grained, fine-

grained) where appropriate [4]. This represents a significant attempt to shed the aforementioned narrow-scope problem which has limited the usefulness of other parallel languages.

Another substantial work in this domain is Data Parallel Haskell [3], which fuses a modern, robust functional programming language (and associated libraries) with an expressive syntax for data-parallel operations similar to that of NESL [2].

1.1 Challenge: Operating Systems

As noted above, traditional operating systems designed for multitasking can often minimize or negate the performance gains sought by parallel application developers. The challenge, therefore, is to develop a tool facilitating a parallel program's uninterrupted access to the raw hardware - this tool must exist alongside the traditional operating system, requiring no modification to the kernel itself. We propose to accomplish this via a virtualization layer, in which a traditional operating system executes as a guest, and is capable of executing parallel applications by spawning *parallel guests*. These guests are also managed by the virtual machine monitor, which maintains full control of system resources, including memory mappings. It is our contention that this capability has the potential to profoundly affect parallel program performance by possibly reducing locality issues, another primary culprit in reducing parallel program performance [10].

This design still requires an operating system kernel to execute within a parallel guest, providing various low-level functions, such as booting and rudimentary device support (for screen printing, for example). In order for a parallel guest to be efficient, it is critical to minimize the footprint of the operating system kernel within the guest, while still providing a reasonable amount of system call and library support.

2 System architecture

Our proposed solution, thus, is to meld high-level application code with a lightweight OS. The end product is a bootable disk image which can be run either directly on a physical machine, in an emulator such as QEMU, or on a virtual machine monitor. Essentially, we have built a toolchain that produces customized operating systems - application code executes within the primary kernel thread of the operating system, and retains kernel-level privileges. Figure 1 shows the components (operating system, libraries, and user application code) that make up this architecture, as well as the compilation dependencies between them.

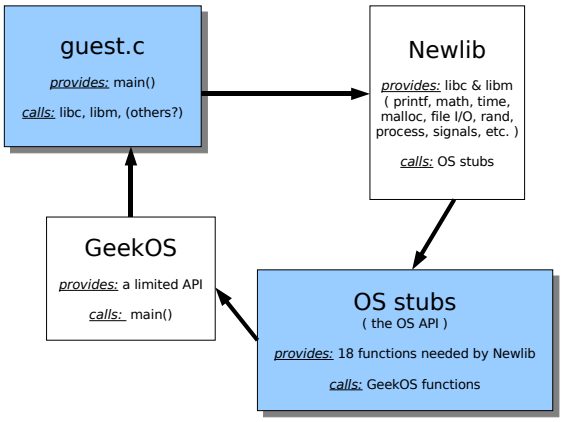

Figure 1: System software components and their dependence relationships

2.1 GeekOS

GeekOS is an educational operating system developed by Hovemeyer et al. [8]. It is written in about 4000 lines of C and 500 lines of assembly code; as far as operating systems go, it is miniscule. As evidenced by the codebase size, its design is very simple and its functionality limited. It is not intended as a Linux or Windows competitor; it primarily provides a basic operating system framework that undergraduate CS students might fully explore, understand, and extend within a semester's time.

GeekOS handles the messy details of

- the x86 boot process
- device drivers for PIC, keyboard, timer, and VGA

- interrupt handling
- kernel memory allocation

but omits implementation of important OS abstractions like user threads, filesystems, IPC, etc.

We chose GeekOS as the foundation of our custom OS toolchain. It is worth mentioning that at least one other OS research project has used GeekOS as a convenient bootable foundation: namely Palacios, a "type-I", non-paravirtualized virtual machine monitor [9].

Our goal has been to support a wide variety of application code, without requiring onerous concessions from the programmer. However, it does seem that at least some concessions are required for the sake of performance. We cannot support something like a full Linux or Windows API while keeping the OS thin; performance and programmability must, to some extent, be balanced. Considering the case of extending GeekOS with application code, the most straightforward solution is to have the programmer write code to be inserted directly into the GeekOS source. That would be painful. As a compromise, we chose to support the standard C library (libc and libm). Beyond that, most of the commonly used C libraries are unsupported, and we do not currently provide POSIX compatibility.

2.2 Newlib

GNU's standard C library (glibc) is used by most Unix-like systems, accomplishing much of its functionality via OS-specific system calls (e.g. I/O and IPC). Though potential guest applications will undoubtedly expect a subset of C library functionality, we avoided porting the glibc library to GeekOS. Because GeekOS has no system call interface (or user-space API), this approach would have required us to create one. Next, we would be faced with porting glibc; this is no easy task since the library is highly optimized for a few important platforms (and is thus very complex).

As an alternative to glibc, we chose to port Newlib to GeekOS. Newlib is an open-source standard C libary implementation initially developed by Red Hat, Inc. [1]. Newlib is a favorite choice of embedded systems software developers [5, 6], because such systems often have no real operating system at all. The beauty of Newlib is that it was designed specifically with portability in mind - a majority of the library is implemented using platform-independent code, and all OS-specific code is delegated to 18 "stub" functions.

A portion of our implementation work involved adding code within these stub functions that targets the GeekOS system-call interface.

3 Implementation

Compilation of a parallel guest is performed in two independent phases:

- The language-specific compiler toolchain (currently restricted to the GHC compiler) is invoked to produce a portable object format file containing parallel application code, as well as a header file declaring C versions of the Haskell module's exported functions.
- A GCC cross-compiler is invoked to link parallel application code with the GeekOS Newlib system call interface.

A cyclic dependency exists between parallel application code and GeekOS kernel code. After spawning a primary kernel thread, our slightly modified variant of GeekOS invokes the entry point into parallel application code (the **main()** function). During the second compilation phase noted above, the linker will expect to locate this symbol within the object file built during phase 1. Similarly, "infrastructure" code (the Haskell runtime in this case) expects to find exported symbols in GeekOS code (the kernel system call interface). In reality, these system calls may either be present directly in GeekOS code or Newlib.

Here we leverage a convenience provided by the GHC compiler - support for the C Foreign Function Interface, which simplifies the means through which a Haskell library might invoke a function in a C library, and vice versa. It is possible to instruct the GHC compiler to produce, in addition to the typical object file containing Haskell code, a C source-

header file combination containing stubs for invoking various Haskell functions defined in the compiled module. For our purposes this provided an easy integration point - a C source file can simply declare a main function, invoking the Haskell application inside this function (note that this does not conflict with GeekOS's own entry point function, which uses Pascal-style casing). For example, consider the following basic Fibonacci sequence implementation in Haskell:

```
module FibHs where
import Foreign.C.Types

fibonacci :: Int -> Int
fibonacci n = fibs !! n
    where
        fibs = 0:1:zipWith (+) fibs (tail fibs)

fibonacci_hs :: CInt -> CInt
fibonacci_hs = fromIntegral . fibonacci . fromIntegral

foreign export ccall fibonacci_hs :: CInt -> CInt
```

Here we define a simple function, and mark that function as callable from the C realm. We invoke the first phase of the compilation process and slightly modify the resulting C source, giving the following:

```
#include "fibonacci_stub.h"
#include <stdio.h>

void main(int argc, char *argv)
{
    int i;
    hs_init(0, argv);
    i = fibonacci_hs(42);
    printf("Fibonacci: %d\n", i);
    hs_exit();
}
```

This main() function will be invoked by GeekOS's primary kernel thread, at which point the user application effectively becomes the guest operating system kernel.

4 Future work

This report has described work that is somewhat incomplete. One area where significant improvement can be made without much effort is in improving support for C guest programs. In particular, rudimentary filesystem support and better tty and/or ncurses support are needed for many of the candidate programs that we attempted, unsuccessfully, to compile into tiny guests.

Though the ultimate goal of the project is to compile any arbitrary application written in a high-level parallel language, the difficulties encountered in attempting to execute simple C programs are significant, because many higher-level programming languages depend on functionality in base C libraries (e.g. Data Parallel Haskell).

4.1 Shortcomings

Though we believe there is tremendous potential in this approach to parallel execution, there remain concerns with the current implementation which must be addressed. Principal among these concerns is the execution of application in privileged kernel mode within the guest operating system - an obviously undesirable scenario, even in the context of a "safe" language such as Haskell which does not allow direct memory access. Currently, the architecture relies on the facilities of the virtual machine monitor to enforce protection, which may yield undesirable performance results.

As a minimalist kernel, GeekOS does not provide features that might be considered essential for supporting a wide range of applications - filesystem support, threading libraries, implementation of the POSIX API, and supplemental C libraries.

4.2 Towards a parallel guest

To date, we have focused our efforts on the successful compilation of an application written in Data Parallel Haskell. Due to resource and time constraints, we have not yet achieved this goal, as the porting of DPH and its underlying compilation framework (the Glasgow Haskell Compiler) has proven a time-consuming task. Haskell offers a sophisticated runtime environment, complete with a garbage collector and user-level threading libraries, which depend on functionality provided by neither GeekOS nor newlib. We also

deem it important to draw definitive conclusions concerning the performance potential of both the GHC runtime system and the DPH parallel libraries, which remains an open research problem, before proceeding with this work.

4.3 Migration from GeekOS

GeekOS has proven to be an excellent starting foundation for this project due to its simplicity. However, GeekOS is not without limitations, most notably

- lack of support for shared memory multiprocessor systems
- lack of filesystem support
- lack of device driver support.

While a minimalist operating system kernel is not without its advantages, as previously discussed, this may become problematic as the toolchain expands its support for broader classes of applications. Other alternatives exists that may provide better support for general-purpose applications, most notably the Kitten kernel. Retaining the "lightweight" moniker while offering a superset of GeekOS's functionality, this alternative has already been explored in related work concerning virtual machine monitor support for SMP systems [9].

4.4 VMM research

Much work is left to be done in the area of virtual machine monitor design and implementation in order to realize the ultimate goal of this project. Interestingly, these problems can be viewed as a generalization of core problems associated with the scalability of parallel architectures and programs today - scheduling and communication.

- The scheduling of multiple parallel guests on a manycore machine still very much remains an open research problem.
- Cooperating guests must be able to communicate efficiently, a concern which has also not yet been addressed by current work.

Perhaps the most attractive option for inter-guest communication involves the development of special "hypercalls", enabling guests to directly call the VMM. This might add flexibility to the design, as the actual implementation details of these hypercalls are left to the VMM and are easily modified (transparently from the perspective of the guest). An alternative approach might employ standard networking protocols on top of virtualized ethernet devices; however, this may introduce both footprint and performance concerns.

Clearly, it remains to be defined what role the virtual machine monitor will play in this architecture, as well as how that role might be implemented. In defining this role, it will be crucial to understand how the overhead associated with a virtual machine monitor will counterbalance the potential performance gains in executing parallel guests.

References

[1] The newlib homepage. `http://sourceware.org/newlib/`.

[2] Blelloch, G. E. Nesl: A nested data-parallel language. `http://portal.acm.org/citation.cfm?id=865063`, 1992.

[3] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming* (New York, NY, USA, 2007), ACM, pp. 10–18.

[4] Fluet, M., Rainey, M., Reppy, J., Shaw, A., and Xiao, Y. Manticore: A heterogeneous parallel language. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming* (New York, NY, USA, 2007), ACM, pp. 37–44.

[5] Gatliff, B. Embedding with GNU: Newlib. `http://www.embedded.com/story/OEG20011220S0058`, dec 2001.

[6] Gatliff, B. Embedding GNU: Newlib, part 2. http://www.embedded.com/story/OEG20020103S0073, jan 2002.

[7] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (New York, NY, USA, May 1991), ACM, pp. 120–132.

[8] Hovemeyer, D., Hollingsworth, J. K., and Bhattacharjee, B. Running on the bare metal with GeekOS. In *In Proc. of the 35th ACM Symposium on Computer Science Education* (2004), pp. 315–319.

[9] Lange, J., and Dinda, P. An introduction to the palacios virtual machine monitor – release 1.0. Tech. Rep. NWU-EECS-08-11, Department of Electrical Engineering and Computer Science, Northwestern University, 2008.

[10] Singh, J. P., Gupta, A., and Levoy, M. Parallel visualization algorithms: Performance and architectural implications. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=299410, July 1994.