Supporting the Performant-Portability Software Stack with the Virtualized Master-Slave Abstraction

Sean Halle Merten Sach Ben Juurlink
Technical University Berlin, Germany
first.last@tu-berlin.de

Abstract

The HotPar 2012 call for papers states that wide uptake of high efficiency parallel architectures "requires new parallel programming paradigms, new methods of application design, new structures for system software, and new models of interaction among applications, compilers, operating systems, and hardware." In short, a new software stack, and a way to organize players in research and industry to supply the pieces of the stack.

A recently proposed candidate for such a software stack[5] relies in part on the assumption that a suitable hardware abstraction exists for the bottom layer of the stack. The abstraction's main purpose is to reduce the effort required in the upper layers. It must reduce the effort of creating language runtimes: by hiding details of synchronization and communication; by regularizing implementation to simplify and encourage reuse across languages; and by reducing the number of runtimes needed by collecting multiple targets below a single interface. It must at the same time enable high performance, by giving the language control over task placement and exposing to the runtime's scheduler the memory hierarchy, communication characteristics, and other major performance-related aspects of the hardware.

In this paper, we show that an abstraction called Virtualized Master-Slave, or VMS [9] satisfies these criteria, and we provide recent measurements to support the case.

1 Motivation

As stated in the call for papers, wide uptake of high efficiency parallel architectures "requires new parallel programming paradigms, new methods of application design, new structures for system software, and new models of interaction among applications, compilers, operating systems, and hardware." Which are element of the software stack, and normally supplied by multiple players in research and industry. The players need to be organized to supply the pieces of the stack.

The goal of the stack is to make parallel programming as productive as sequential programming, and to make it as portable onto new generations of hardware as sequential code.

Reduced cost of parallel software is one major benefit of such a stack. Part of the cost reduction comes from performant portability. This means code is written once then run performantly across hardware targets, including unknown future architectures.

A recent proposal for achieving this, named PStack[5], calls for a software stack having a layer of languages (toolchains) at the top, a layer of runtimes below that, and a hardware abstraction layer at the bottom.

It is this bottom layer that we focus on in this paper. We begin by giving context for the bottom layer with more information about the software stack, in Section 2. We then explore the requirements for the layers in in Section 3, and show how VMS satisfies the requirements in Section 4. We then move to results, giving our experimental setup in Section 5, and measurements in Section 6. In Section 7 we tie the elements of the paper together in the conclusion.

2 Context: PStack

Many projects are attempting portability [10, 2, 4, 3, 1]. PStack differentiates itself in three ways: 1) it's goal is wider than most: (nearly) all-languages to (nearly) all hardware 2) It's approach is to *organize* – industry supplies the pieces of the solution, while PStack itself only provides the interfaces and scaffolding, along with the seed of a solution to start 3) PStack has unique approaches for the application interface and the hardware interface that fill fundamental needs.

The general philosophy is that portability involves too much effort to be solved by a single group. Instead, an industry-wide effort is needed, where each player provides one small piece of the solution. This, though, requires some way to organize it all, and modularize the pieces.

PStack addresses this by defining a number of interfaces, and providing tools to manage specialization. These result in a simple, decoupled process for adding new solution pieces. So, the solution can grow at its own pace, accumulating the efforts of many.

2.1 PStack elements

As seen in Figure 1, at the top, a standard set of information is defined, which must be gathered from the application. Current languages don't capture all the required

information. So PStack defines a set of constructs to be added to a language to fill its gaps. The added constructs are denoted "+P" appended to the language name.

In the middle, standard runtimes require too much effort to create, and discourage reusing schedulers across languages. So PStack defines a hardware abstraction that removes as much as possible from the runtime, including concurrency in the runtime itself. The abstraction makes the runtimes all have similar structure, which simplifies reuse of complex scheduler code among languages.

At the bottom, performance of the runtime itself requires intense low-level hand-tuning and debugging. This is captured inside the implementation of the abstraction. It is done once for each hardware target, then reused across the runtimes from all languages. So the intense hand-tuning is taken out of the runtimes, in the middle layer, while it benefits all languages and hence applications in the higher layers.

2.2 How VMS influences the stack

VMS was chosen as the abstraction in the bottom layer. However, VMS affects multiple interfaces and layers of the stack. At the top, it determines the way parallel constructs are embedded into base languages, and how custom-syntax languages generate their runtime-interactions. Next, between the top and middle, VMS defines the interface for the language layer to talk to the runtime layer. Then within the middle layer, VMS defines two standard function prototypes, so that a runtime consists of implementations of just these two functions. Between middle and bottom, VMS defines a number of services that runtimes in the middle can call, and also defines the interaction between the VMS-implementation and the two runtime functions.

3 Requirements

Although the paper focuses on the bottom abstraction, its requirements are influenced by the layers above. So we present a full picture of the requirements in the stack, to give a complete picture for the bottom abstraction.

3.1 Top Layer: Language Requirements

The languages must be designed to capture all information required to specialize the source for high performance on any target hardware. A computation model, called The Holistic Model[**?**], suggests that such a canonical set of information exists.

PStack proposes to develop the constructs that gather the canonical information set, where some constructs are in the form of specialization helpers such as task-resizers and layout modifiers. The application implements the specialization helpers, thereby encoding information about data structures and how to manipulate them. The seeds of such an approach were laid with work on DKU[7], which demonstrated the success of task-resizing constructs.

PStack also calls for the use of the BLIS[6] approach for managing multiple toolchains, where each toolchain specializes to a different target. The management covers the install process, during which the correct toolchain output is paired to the installation target. Further specialization can thus be naturally added during installation, when exact hardware details are known. If required, runtime tuning and optimization also fit naturally within the approach.

3.2 Middle Layer: Runtime Requirements

Below the top layer, a collection of runtime systems acts as a sort of cross-bar switch, connecting the languages above to the hardware abstractions below. Such a "cross-bar" switch made up of runtimes implies a large number of runtimes.

To be practical, the number of runtimes must be reduced; the effort of creating one must be reduced; and reuse of sophisticated runtime code must be encouraged.

3.3 Bottom Layer: Abstraction Requirements

The primary purpose of the bottom abstraction is to reduce the effort of creating the runtime layer.

- The abstraction must hide details, making multiple hardware targets present the same interface and use a common runtime.
- The abstraction must hide low-level tuning of the runtime itself, like synchronization-related tuning.
- The asbstraction must provide common services, such as handling internal synchronization of the runtime, creation of tasks, communication, etc.
- The abstraction must create uniform patterns for runtime implementation, making reuse between runtimes more practical and reducing the effort of making multiple runtimes.

However, the abstraction must not hide *application*-performance-critical information from the runtime, which holds the scheduler that decides when tasks become ready and where to execute them. The scheduling choices need to know the communication paths and memory pools in the hardware, along with latency, bandwidth, capacity and computation rate.

A single abstraction can't both hide details and expose those required by the runtimes to attain high *application* performance. Instead, PStack calls for a family of abstractions, one for each major type of architecture, including a "hierarchy" abstraction used to glue together heterogeneous hardware. In each, only the details critical to application performance are exposed to the scheduler in the runtime, thus keeping the number of abstrac-

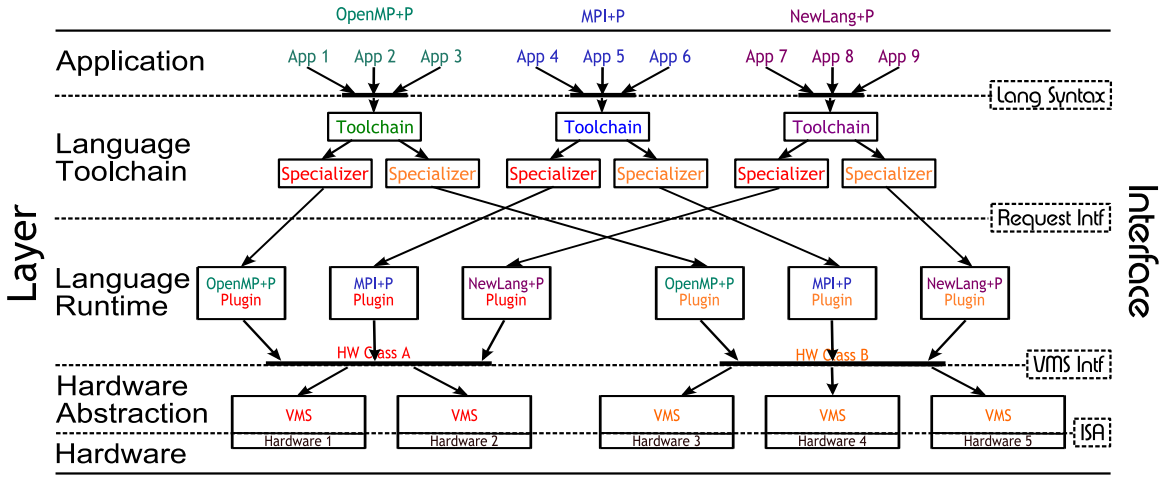

Figure 1: Depiction of PStack, with layers named on the left, and interfaces between layers named on the right. At the top are toolchains plus specializers, in the middle are runtimes connecting languages to hardware, and below that are hardware abstractions that collect similar hardware below a single interface and simplify runtime implementation.

tions needed manageably small, on the order of tens in total.

4 Relating VMS Details to Requirements

Given the requirements, how does VMS meet them? We given more detail on VMS, at each place it affects the stack, and show how the details satisfy the requirements.

4.1 Top-layer

With VMS, a language is implemented as either a collection of wrapper-library calls embedded into a base language, or as custom syntax. The wrapper-lib functions call a primitive supplied by VMS that suspends the virtual-processor animating the call, and sends a request to the runtime. This same VMS primitive is also used to implement custom syntax, inside the compiler. Thus, the VMS primitive is the means for the language layer to interact with the runtime layer.

VMS is invisible to the application, only language constructs are visible – either wrapper-library calls or custom syntax. From the application-programmer point of view, even an embedded parallelism construct looks like a function call, albeit the data-structure of the virtual-processor animating the code has to be passed as a parameter to the wrapper-lib call.

4.2 Interface from top to middle

The interface between application-executable and language-runtime is fixed, as the VMS-primitive that sends a request to the runtime. Even though PStack allows executables to be modified during installation or even runtime, via BLIS management of auto-tuners, multi-stage compilers, or binary re-writers, the VMS-primitive still must be used for the executable to interact with the runtime.

Such a standard interaction mechanism serves not only to modularize the stack, cleanly separating runtime from toolchain, but also to decouple executable from VMS implementation. The VMS primitive is naturally a custom instruction, but can also be, a trap to the OS, a message sent on a port, or a function call – given appropriate executable modification under BLIS.

4.3 Middle layer

VMS causes the middle-layer portion of a runtime to be implemented as two functions. The first is the request-handler, which is the part of a scheduler that handles constraints. It determines which work units (tasks) are ready to be animated (executed). The other function, sched-assigner, assigns ready work to hardware. This provides uniform patterns for the runtimes.

When a request is ready for the runtime, VMS calls the request-handler function, and when hardware is free for work, VMS calls the scheduler-assign function. Thus, the language portion of the runtime is passive.

By keeping control-flow inside VMS, the language-supplied portion of the runtime is simplified. Control flow includes any concurrency, and so is inside the VMS-implementation. Hence, the language-supplied runtime functions are sequential code, even though they implement the *semantics* of language-level synchronization constructs. This simplifies runtime implementation.

This structure is also the reason VMS encourages reuse of scheduler code. Scheduling is sub-divided into distinct modules: constraint-management (IE enforcing dependencies); and assigning work to resources. The assignment module is especially straight-forward to share between languages.

Because application performance is most strongly influenced by communication within the hardware, the assignment module is critical. For high performance, it also tends to be complex. Thus, simple reuse of it is a significant benefit.

4.4 Interface from middle to bottom

VMS's plugin API is the interface between the runtime and the bottom abstraction-implementation. The API has calls to register language-supplied runtime functions with the bottom abstraction, as well as support services.

Reduction of the number of runtimes is accomplished this way. Hardware targets with similar structure present the same interface, requiring only one runtime.

Only structural elements that affect assignment choices are exposed in the API. For example, memory hierarchy is exposed as a VMS-defined data-structure made available to the sched-assign function. The details in the data convey the connectivity, communication, and sizes, which the assigner may use to optimize choices.

4.5 Bottom layer

The bottom layer consists of implementations of the VMS API and VMS primitives used in the upper levels, as well as the control-flow of the runtimes. Each hardware platform has its own implementation, allowing low-level hand-tweaking. This effort is performed once per hardware target, so is amortized across applications. Pulling this tuning below the interface also simplifies the runtime-portion in the middle layer.

5 Experimental Setup

The experiments to measure VMS overhead were run on three machines: a one-socket 2 core 3GHz workstation ("1x2"), a one-socket 4 core SandyBridge 3.3GHz workstation ("1x4"), and a four-socket by 10 core each Westmere EX 2.4GHz server ("4x10").

The code consists of two loops: the innermost is a single task, while the outer repeats that task a number of times. The inner does throw-away work entirely within registers, where the number of iterations sets the amount of work in the task. After the inner completes, a synchronization is performed, which pairs each task to a sync operation. The outer then repeats the sequence of task-then-sync a large number of times to gain statistical accuracy and dominate any other sources of overhead.

Two versions of the code were written: one that used pthread, a second that used a VMS-implemented equivalent called Vthread. Both have the same semantics, differing only in the implementation of scheduling triggered by the construct. Hence, any difference in execution time is due to the difference in scheduling overhead.

6 Results

The new experimental results given in this paper focus on the overhead of the runtime, with the goal of showing that a language based on VMS enjoys low overhead compared to standard pthreads. We illustrate the amount of overhead by plotting a curve whose shape is determined by the overhead.

The curve compares total CPU time to just work time. The difference is the overhead of scheduling, which consists of: switching from application to scheduler; updating the sync-construct state; choosing a new thread to schedule; and deciding on which core to re-animate it.

The ratio of total CPU time to work time gets larger as the overhead increases, raising overhead's percent of the total. When the ratio is exactly 2, the work time exactly equals the overhead. Larger ratio indicates overhead dominates, smaller indicates work dominates.

Hence, to find the size of the overhead, find the size of task where the work in the task exactly equals the overhead of scheduling the task. To do this, we plot the ratio on the y axis and single-task-time on x axis. When the ratio equals 2, the cycles of work in the task equals the overhead of scheduling the task. So the overhead can be read off the graph, as the task-size at the y=2 point.

6.1 Performance Results

We executed on each of three machines. On a given machine, we first executed the pthread version, then the Vthread version, with a variety of numbers of threads. Varying the number of threads shows the effect on scheduling time. For a given machine, both sets of curves are plotted on the same graph, to make direct comparison easy.

Figure 2 shows results for the 1x2 machine. The curves for Vthread cluster together in the lower-left, indicating that overhead is smaller than for pthread. The tight clustering means that overhead remains constant as the number of threads is increased.

The values for overhead per task is read off the graph by finding where the curve crosses $y = 2$. This shows that Vthread has around 700 cycles of overhead, while pthread starts at 3800 for 8 threads, goes up to 8200 for 32 threads, and then into the tens of thousands for 128 threads. Not shown is the curve for 512 threads, which has more than 100,000 cycles of overhead.

Figure 3 shows similar characteristics on the 1x4 SandyBridge machine.

However, things change dramatically on the 4 socket by 10 core-each Westmere machine, seen in Figure 4. Here, inter-socket communication dominates, and VMS gains orders of magnitude advantage. For one thread per hardware context, Vthread's overhead is around 1500 cycles, while pthread starts at around 50,000 and goes up from there.

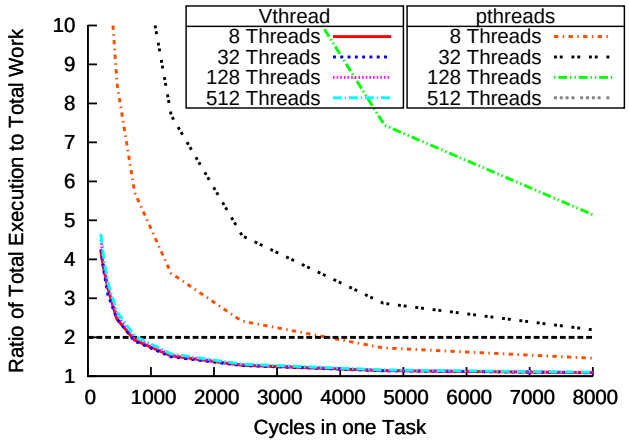

Figure 2: $\frac{ExecutionTime}{WorkTime}$ vs $TaskTime$ on the 1x2 machine. It shows results for Vthread and pthread on the same axes, for 8 through 512 threads. The Vthread curves cluster, appearing as the bottom-most, while the pthread curves for 8, 32, and 128 are above it. The results for pthread with 512 threads land outside the plot.

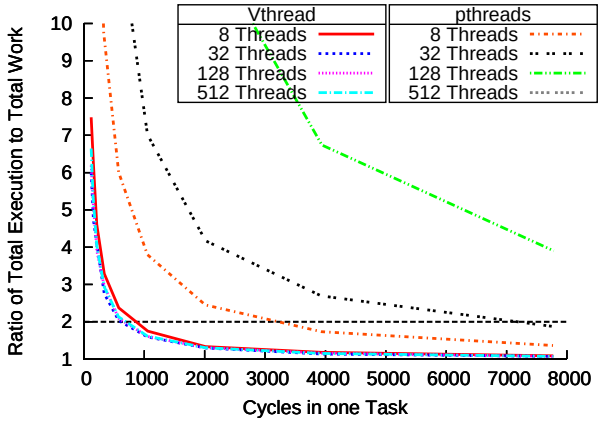

Figure 3: $\frac{ExecutionTime}{WorkTime}$ vs $TaskTime$ on the 1x4 machine. The results are similar to Figure 2.

The implementation of VMS is different on this machine than the single-socket ones, and demonstrates the effectiveness of pulling hardware details below the abstraction.

When using the single-socket implementation on the 4x10, the large number of cores and inter-socket communication times causes excessive contention. We solved the problem for the 4x10 machine with an increasing-random-backoff approach. It reduces overhead by an order of magnitude on the 4x10.

Without an abstraction like VMS, the language implementers would have to discover and solve such problems separately for each language on each machine. Because this required several weeks, the use of advanced tools, and detailed knowledge of the hardware, the savings for

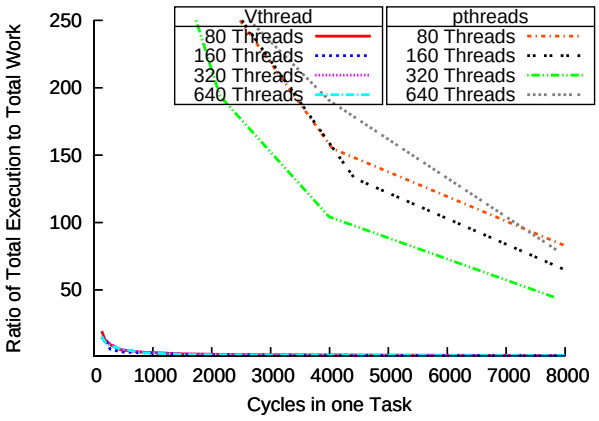

Figure 4: $\frac{ExecutionTime}{WorkTime}$ vs $TaskTime$ on the 4x10 machine. The Vthread results are difficult to see, at the bottom of the plot, while the pthread results appear in the middle. The runs start at 80 threads, which is the number of hardware contexts in the machine.

the language-runtime implementers is significant. This is evidence of VMS's ability to reduce middle-layer runtime implementation effort.

6.2 Implementation Time Results

As seen in a previous paper on VMS[8], it makes runtime implementation quick and easy. The results are reprinted in Table 1 to support the claim VMS meets the requirement of reducing runtime implementation effort.

From previous experience and informal discussions with others, equivalently low-overhead tuned runtimes would take several months. Similar time is also expected to learn the code of a pre-existing multi-threaded highly tuned runtime, then modify, debug and re-tune it.

Table 1: Person-days to design, code, and test each of three sets of parallelism constructs. L.O.C. is lines of (original) C code, excluding libraries and comments.

	SSR	Vthread	VCilk
Design	4	1	0.5
Code	2	0.5	0.5
Test	1	0.5	0.5
L.O.C.	470	290	310

7 Conclusion

We showed that using VMS as the bottom hardware abstraction in a software stack pulls low-level tuning out of the runtimes, reduces the number of runtime implementations, and encourages reuse of scheduler assignment code across languages.

References

[1] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. *First Workshop on Programmable Models for Emerging Architecture at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[2] Intel Corp. TBB home page. http://www.threadingbuildingblocks.org.

[3] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, 2006.

[4] Kronos Group. OpenCL home page. http://www.khronos.org/opencl.

[5] Sean Halle. PStack home page, 2012. http://pstack.sourceforge.net.

[6] Sean Halle and Albert Cohen. BLIS website, November 2008. http://blisplatform.sourceforge.net.

[7] Sean Halle and Albert Cohen. DKU website, November 2008. http://dku.sourceforge.net.

[8] Sean Halle and Albert Cohen. A mutable hardware abstraction to replace threads. *24th International Workshop on Languages and Compilers for Parallel Languages (LCPC11)*, 2011.

[9] Sean Halle, Merten Sach, Ben Juurlink, and Albert Cohen. VMS home page, 2010. http://virtualizedmasterslave.org.

[10] OpenMP organization. OpenMP home page. http://www.openmp.org.