Support of Collective Effort Towards Performance Portability

Sean Halle, Albert Cohen

Published in Usenix's HotPar 2011

Support of Collective Effort Towards Performance Portability

by Sean Halle and Albert Cohen

Email: first.last@inria.fr

Abstract

Performance portability, in the sense that a single source can run with good performance across a wide variation of parallel hardware platforms, is strongly desired by industry and actively being researched. However, evidence is mounting that performance portability cannot be realized at just the toolchain level, or just at the runtime level or just at the hardware abstraction level.

This is a position paper, making a suggestion for how the groups involved can more efficiently solve the performance portability problem together. We don't propose a solution, at all, but rather a support system for the players to self organize and collectively find one. The support system is based on a new *extendable* virtualization mechanism called VMS (Virtualized Master-Slave), that fulfills the needs of an organizing principle, and provides focus that may increase research efficiency. The difficult work will be the on-going research efforts on parallel language design, compilers, source-to-source transform tools, binary optimization, run-time schedulers, and hardware support for parallelism. Although it doesn't in itself solve the problem, such an organizing principle may be a valuable step towards a solution – the problem may be too complex and require cooperation of too many real-world entities for a single-entity solution.

We briefly review VMS, and illustrate how it could be used to give rise to an eco-system in which performance portability is collectively realized. To support the suggestion, we give measurements of the time to implement three parallelism-construct libraries, and performance numbers for them, along with measurements of the basic overhead of VMS.

1 Overview and Motivation

Evidence is mounting that one-stop solutions to performance portability fail to address critical real-world patterns. For example, attempting to place the full specialization into the toolchain centralizes the specialization effort in a single organization. One route is PetaBricks style [5] which places specialization into a single tool that injects an adaptable runtime. Another is BLIS style [20] which automates re-compilation for new targets and automates distribution of multiple binaries – to do this for thousands of different software development entities requires centralization. Either way, such a centralized approach concentrates control, creates a choke-point slowing the pace of innovation, and has serious unresolved technical challenges.

Pure runtime based approaches [19] imply a single binary, with only the runtime dynamic-library changing across targets. This frees hardware manufacturers to independently deploy runtime libraries. However, for performance they must take advantage of language-specific constructs. This forces a separate runtime for each language. Without a means to simplify the creation of such runtimes, and reuse effort across hardware, this represents significant development effort.

Finally, hardware abstraction based approaches, such as JITs [7][27] or VMMs [26], place the work of specializing into the virtual machine, which makes reuse difficult, forcing rewrite of JIT internals for each hardware platform, for each language. This software cost is an issue in the embedded space where new hardware is introduced often and has a limited market size to amortize the software cost. In addition, this one-stop approach requires a different JIT for each language, because it has to recognize language-specific features to specialize – or else it fails to achieve good performance. This requires extensive work, making domain-specific languages time-consuming and difficult to develop, and the multiple JITs logistically awkward.

In practice, most approaches combine at least two in a limited way. We take the position in this paper that a support system is needed that operates at all three levels: the language design plus toolchain level; the runtime system level; and the hardware abstraction level. Sequoia [12] does this, in a limited way. We propose a more general approach that supports (multiple) languages without restricting them, has more freedom to specialize the runtime, and reduces the effort to add new target hardware. It takes advantage of VMS (Virtualized Master-Slave) [18][17], which provides pieces for each level, acting as an organizing principle.

Section 2 reviews VMS. Section 3 reviews performance portability fundamentals. Section 4 illustrates using VMS to support an eco-system. Section 5 gives details of a VMS implementation, and 6 gives VMS performance. Section 7 concludes.

2 Overview of VMS

On nomenclature, in this paper we define *task* as a 3-tuple: <animation event, collection of code animated, collection of information the code is animated upon>. The phrase "we create a task" means we create a combination of code plus data with the intent to animate.

Virtualized Master Slave (VMS) is an *extensible* virtualization mechanism that replaces Threads. Each language has its own scheduler, along with a definition of parallelism constructs, such as publish-subscribe channel, or send-receive, or spawn-sync. This scheduler plus parallelism construct implementation are plugged in to VMS, and, together, complete the runtime for the language.

The plugin is separately loaded onto the hardware as a dynamic library or device driver. This makes the plugin a new component in the software stack.

The VMS model can be implemented as a user-space library, or existing OS kernels can be converted to the VMS model by opening up their scheduler, exposing the VMS plugin interface. The parallelism constructs are implemented using sequential algorithms, reducing implementation time for parallelism semantics to a matter of days. The plugged-in scheduler gives the language control over assigning work to physical cores. On multi-core shared-memory machines, the run-time overhead is low, around a few hundred nano-seconds per concurrency operation.

VMS isn't a language, but rather *supports* the creation of language runtimes. Examples could include: Actors [21][3]; Components [23]; process calculi like CSP [22] and Pi-calculus [24]; and coordination languages like Linda [15]. Systems like TBB[9], OpenMP[25], and Sequoia may also be implemented with VMS, as well as the threading portion of Java, and even pthreads.

3 Specialization

Portable performance is achieved by specializing the source code to the hardware. For traditional sequential source on sequential processors, the specialization was the translation to machine code and optimization that took place in the compiler.

This one-step specialization was sufficient because most of the performance portability was provided by using micro-architecture hardware techniques that exploited instruction-level parallelism underneath a standard instruction-set abstraction (plus faster clock).

We conjecture that the reason such a hardware-based approach to performance portability has failed for parallel hardware is that larger granularity parallelism is tied to the language constructs and to application constructs (whereas instruction-level parallelism is tied only to the machine-instruction-set constructs). Hence, information about the language patterns and application patterns must be available in order to exploit these larger granularities.

Following this intuition, we propose that a system for parallel performance portability have a means to identify language and application constructs and package this information into a standard format. Also, a hardware abstraction should be provided that accepts such information and uses it to make high-quality decisions about task creation, sizing, and placement.

3.1 Three-step specialization

VMS fits this proposal by making specialization happen in three steps, as illustrated in Figure 1.

First, in the top box, the toolchain extracts task information useful to the scheduler and packages it into the binary. This specializes the source to the plugin's interface. Second, in the middle, the scheduler in the plugin retrieves the info and uses it to make scheduling decisions. This is specializing the binary to the hardware abstraction. Third, at the bottom, the VMS-core implementation hides hardware details behind the interface. This is specializing the hardware abstraction to the hardware.

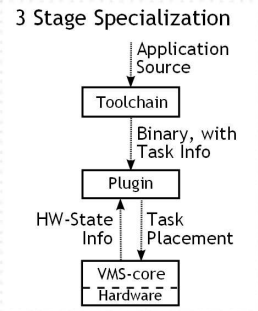

Figure 1. Specialization occurs in three places.

The combination of plugin plus VMS has the same function that the instruction set had back in the sequential days – it provides a standard hardware abstraction. VMS has the advantage that the abstraction can be chosen separately for each binary, by choosing the plugin.

Isolating parallelism in the toolchain: Focusing on the toolchain, we propose breaking it into two sections: one for parallelism, the other for sequential. The parallelism portion extracts the task and language information needed to make high quality scheduling decisions. The sequential specializes individual functions to the sequential cores. The parallelism portion remains constant across hardware, only the sequential portion changes when the instruction-set of the target hardware changes (in effect, this moves a portion of the compiler into an install-time, or load-time, or run-time component that completes binary specialization).

One possible way to achieve this is to make the parallel portion transform the original source to C code, with embedded calls to the parallelism constructs. Also during this source-to-C-plus-lib-call transform, the task information is packaged into functions in some way. The resulting C-plus-lib-calls source is then compiled with a sequential C compiler to make a binary, as depicted in Figure 2.

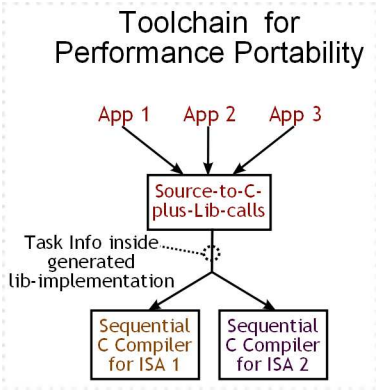

Figure 2.
The toolchain is split into parallel and sequential portions. The parallel portion generates the bodies of information-carrying functions that the plugin later calls to retrieve the info.

Meanwhile, the plugins for that language know the names of the library functions the task information has been packaged into. Hence, when the binary is linked to a plugin at load time, the task-info functions within the binary become available to the plugin. They are called during the run to extract the information.

This scheme allows unmodified sequential compilers to be used to pass the information along, inside of a standard binary format. It also separates the parallel and sequential portions of the tool chain cleanly, so only the relatively simple sequential C compiler changes with hardware. This scheme also leaves the language designers in charge of deciding the nature and definition of the information carrying functions.

These properties enable reuse of the same parallel portion of the toolchain across chips, which is especially valuable in the embedded market. With the inclusion of real-time aspects, such as latency bounds and deadlines, this could dramatically speed up time to market and reduce cost of introduction of new embedded chips.

Focus Area: This part of the proposal may run into difficulties in practice. The details of architectures like vector units and GPUs are too fine-grain for typical run-time scheduling, and may require multi-versioned binaries[13][8] or binary optimizers[11][28] or PetaBricks style adaptability. It will require extensive toolchain research by many groups to either solve this or give reasonable evidence that it's not practical.

What task information to extract: This begs the question: what kind of task information is sufficient for performance portability across the array of foreseeable parallel architectures? Fortunately, the support system doesn't define this, but leaves it free to evolve as research progresses, enabling an upgrade path.

We propose three kinds of information will be sufficient: 1) manipulators, which are able to modify the size of tasks, choose among alternative binary versions, and change the data layout and access pattern (such as auto-tuners injected into the binary and invoked by plugin), 2) information about the tasks such as communication with other tasks (for data affinity), preferred core type, data footprint, and predicted execution time, and 3) real-world constraints that relate to the tasks, such as deadlines, maximum latency for data to pass from one point in the computation to another, and quality related information. Again, it is up to the language and plugin to agree on what data is extracted, passed, and then used for scheduling. Meanwhile, VMS must provide the scheduler with the services needed to make use of that data.

As research progresses, additional types of information may prove needed, so flexibility from the plugin system will be key. The plugin plus VMS implementation are a hardware abstraction – the parallel equivalent of what the instruction set used to be for sequential processors, but now defined in software.

Info exposed by VMS to plugin: This, finally, begs the question: what kind of information and services must VMS provide to the plugin? It must expose the structure of the hardware that matters most to performance (computation, energy, and real-world related performance).

We believe that the type(s) of cores, the pools of memory and communication between them are the most important features for parallel performance. However, it should only expose the portions of memory affectable by a runtime scheduler (the register set usage of a sequential processor or vector processor is fixed by the binary and so not exposed).

On the nature of communication, we believe that the scheduler can safely model any network with a topology-independent statistical model, without undue loss of performance [2][4]. This leaves the main feature as coherent memory vs distributed, which determines whether communication takes place by shared variables in the code vs whether it needs explicit action.

Following this, we propose that for the purposes of scheduling, most parallel hardware will be adequately modeled by a simplified hierarchical graph. Nodes are of type: 1) physical memory array, 2) processor pipeline, 3) internally-scheduled sub-graph. Communication links are chosen from: 1) automated movement (ex: due to coherence mechanism) 2) explicit movement. The links are statistical models of how well the physical network moves data.

The internally-scheduled sub-graph is the key feature. It allows an entire GPU, for example, to be treated as one run-time schedulable node as StarPU [6] does. It also allows a hierarchy of plugins to exist for complex hardware. Higher level plugins schedule large tasks to sub-graphs, in which the tasks are again divided (using manipulators packaged by the toolchain, such as demonstrated by PetaBricks and DKU[19][1]). Techniques such as Hierarchically Tiled Arrays [16] and the loop manipulation features of the X Language [10] will also facilitate such hierarchical scheduling.

We suggest that most parallel architectures will eventually fall into a small number of groups. All targets in a group have similar graphs. This conveys sufficient structure to efficiently schedule any target in the group, without exposing chip-specific details.

Focus Area: This will be difficult research (some of which is currently in progress) developing low-overhead schedulers that have a single binary-interface, but efficiently handle a range of related hardware graphs. An example would be a single plugin whose scheduler is efficient on various multi-core systems with GPU accelerators, regardless of which particular multi-core and GPU chips are present, possibly adapting job-scheduling approaches [14].

4 Eco-System

Figure 4 depicts how many real-world entities might interact to supply the various pieces. At the top, independent software developers write applications, in a

variety of languages. Each language is defined by a research group, along with its own format for conveying task-related info.

The plugins, in the middle, act as a cross-bar switch, connecting the binaries to the hardware abstractions. They are separately distributed and loaded onto the hardware, and separately written. The middle will fill out as research teams discover scheduling techniques for various groupings (classes) of hardware. Each implements a plugin for a hardware class, which accepts a particular language's format. For unusual hardware designs, the manufacturer supplies their own plugin for the popular languages, thereby taking advantage of the existing application binaries (non-standard instruction sets also need an install-time translator).

At the bottom, the VMS-core implementations standardize the hardware. They are mainly supplied by the hardware manufacturers.

The minimal software needed for a new HW platform is the VMS-core abstraction, sequential C compiler, and a bare-bones OS (and possibly a binary-optimizer). Existing applications are adapted via the plugins for the hardware class and the abstraction.

We expect that a small number of HW classes will quickly come to dominate, which will encourage later HW development to fit within the dominant classes. As a result, a standard set of plugins, and sequential C compilers will emerge, allowing software developers to perform a single compilation pass. For fine tuning of *sequential* compilation choices (or optimization of fine-grain hardware like SIMD GPUs), we expect install-time binary re-writers or run-time binary optimizers.

The end-result is that no centralized control is needed. Language designers innovate, inventing new parallelism constructs, and need as little as just a source-to-C translator to reach all the standard hardware platforms. Likewise, hardware manufacturers innovate freely, needing minimal software development for a new chip to enjoy access to existing development tools and applications.

Note that new hardware still has the option of aggressive binary optimizers. In this scheme they have richer application information available than normal due to the task information bundled into the binary.

The set of plugins is the key to this portability, in combination with funneling many applications to the same parallelism information at the top, and funneling many hardware platforms to similar graphs at the bottom.

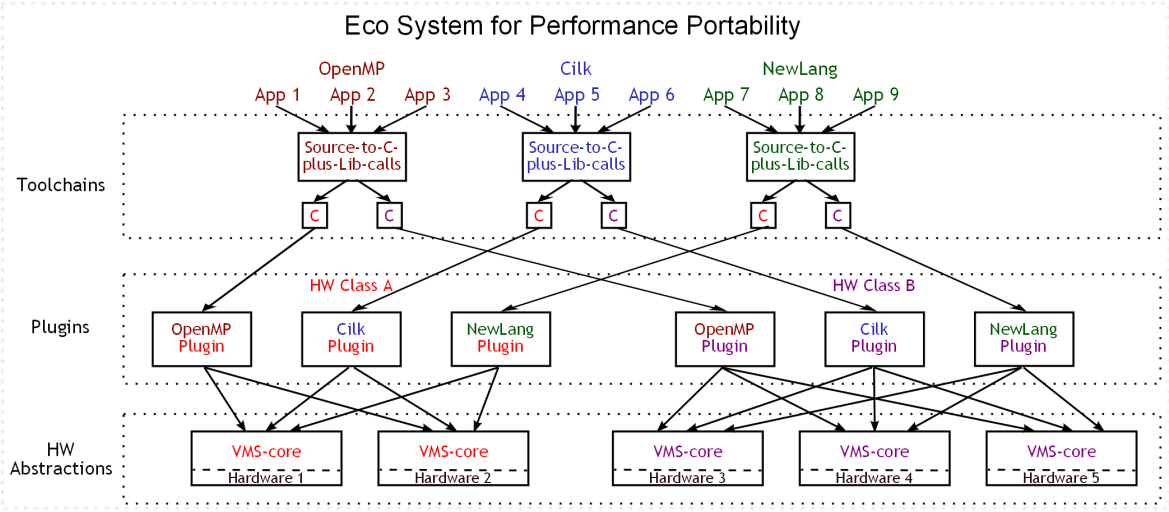

Figure 3. Eco-system is composed of toolchains, plugins, and HW abstractions. Each element, such as a particular plugin or sequential C compiler, is supplied by a different physical-world entity, such as a company or a research group. Elements related to a particular language are all shown in the same color, while elements related to the same hardware class are also shown in the same color. The plugins combine a language color with a hardware class color because they depend on both. As can be seen by the coloring, the toolchain for a language is independent of HW except for the sequential C compilers.

5 Internal Workings of VMS

To give greater understanding of VMS, we describe the internal working of our implementation for shared-memory multi-core hardware. It has a master virtual-processor (VP) on each core, into which the plugins are inserted, and a "core-controller" that handles transferring the physical-core between animating the master VP and animating the slave VPs. The plugin functions are called by the master VP and control which slave goes to which core (more detail shortly). All application code is animated by slave VPs. When app-code invokes a parallelism construct, the slave VP first attaches to itself a request representing the parallelism construct (and destined for the plugin in the master VP) then suspends. We next review how this request

reaches the plugin's parallelism-construct implementation, called the *request handler*, by walking through the steps VMS takes during operation.

Steps of Operation: The steps of operation are numbered in Figure 4. Taking them in order, 1) `master_loop` scans the scheduling slots to see which ones' slaves have suspended since the previous scan. 2) It hands these to the request handler plugged-in. 3) The data in the request causes the request handler to manipulate data structures within the shared semantic state. These structures hold all the slaves currently suspended. 4) Requests cause slaves to be moved to the ready container (for the plugin shown, this is one queue on each physical core – semantic constructs and work-stealing determine which core a slave is assigned to). 5) During the scan, the `master_loop` additionally looks for empty slots. For each, it calls the scheduler plug-in function, which returns a slave (this plugin just pops the Q). 6) The `master_loop` then places the slave VP's pointer into the scheduling slot, making it available to the `core_loop`. 7) When done with the scan, `masterVP` suspends, switching animation back to the `core_loop`. 8) `core_loop` takes slave VPs out of the slots, then 9) switches animation to them. 10) When a slave self-suspends (due to app-code), animation returns to the `core_loop`, which picks another, until 11) all slots are empty and the `core_loop` again switches animation to the `masterVP`.

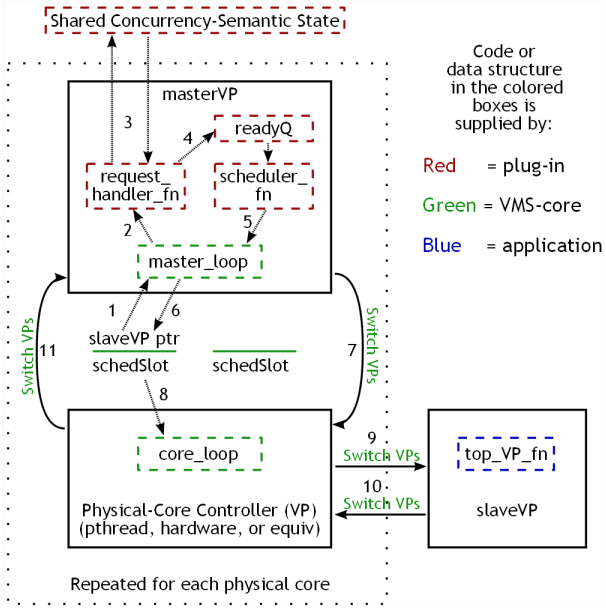

Figure 4. Internal elements of VMS implementation

Sequential implementation of parallel constructs: In this particular implementation we use a central `masterLock` to ensure that only one core's `masterVP` can be active at any time. This guarantees non-overlap of `masterVP`s on different cores, allowing the plugins to use sequential algorithms.

6 VMS Measurements

Setup: We implemented blocked dense matrix multiply and ran on a Core2Quad 2.4Ghz chip.

Implementation-Time: As shown in Table 1, time to implement the three parallel libraries averages 2 days. As an example of productivity, adding nested transactions, parallel singleton, and atomic function-execution to SSR required a single afternoon, totaling less than 100 lines of C code.

	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

Table 1. Person-days to design, code, and test each parallelism library, in the order attempted. L.O.C. is lines of (original) code, excluding libraries and comments.

Execution Performance: Performance of VMS is seen in Table 2. The code has not been designed for speed, but rather to be easy to understand and modify. In particular, the schedulers are simple queues with no optimization for performance.

		comp only	comp +mem
VMS Only:			
`master_loop`		91	110
switch VPs		77	130
(malloc)		160	2300
(create VP)		540	3800
Master lock			250
Library:			
SSR	plugin – concur	190	540
	plugin – all	530	2200
Vthread	plugin – concur	66	710
	plugin – all	180	1500
VCilk	plugin – concur	65	260
	plugin – all	330	1800

Table 2. Cycles of overhead, per scheduled slave. "comp only" is perfect memory, "comp + mem" is actual cycles. "Plugin-concur" only concurrency requests, "plugin-all" includes create and malloc requests. Two significant digits due to variability.

Head to Head: Comparing our implementation of the `spawn` and `sync` constructs against the distributed version of Cilk, the same application code has similar performance. VCilk does 23% worse on large matrices that run for several seconds, but 210% better on small matrices requiring only milliseconds. Versus pthreads, our VMS based implementation has more than an order of magnitude better overhead per invocation of mutex or condition variable functionality.

7 Conclusion

This is a position paper, merely a suggestion for a support system for cooperatively achieving performance portability. The key pattern is the funneling of many applications to the same parallelism information at the top, the funneling of many hardware platforms to the same abstraction at the bottom, and the set of plugins that connect the two ends.

Bibliography

[1] DKU-ized Deblocking Filter code. http://dku.svn.sourceforge.net/viewvc/dku/branches/DKU_C__Deblocking__orig/.

[2] Anant Agarwal. Limits on interconnection network performance. *Parallel and Distributed Systems, IEEE Transactions on*, 2:398 – 412, 1991.

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[4] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in global computing systems. *INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS*, 14:268–279, 2000.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, 2009.

[6] CÃľdric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-AndrÃľ Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing*, Lecture Notes in Computer Science, pages 863–874. 2009.

[7] 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.

[8] Charles Consel. A general approach for run-time specialization and its application to c. pages 145–156. ACM Press, 1996.

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

[10] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, MarÃŋa GarzarÃąn, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 136–151. 2006.

[11] E Duesterwald. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93:436 – 448, 2005.

[12] 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.

[13] Grigori Fursin, Albert Cohen, Michael OâĂŹBoyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In Tom Conte, Nacho Navarro, Wen-mei Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science, pages 29–46. Springer Berlin / Heidelberg, 2005.

[14] JÃűrn Gehring and Friedhelm Ramme. Architecture-independent request-scheduling with tight waiting-time estimations. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 65–88. 1996.

[15] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[16] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 111–122, 2008.

[17] Sean Halle. VMS home page. http://vmsexemodel.sourceforge.net.

[18] Sean Halle and Albert Cohen. An extensible virtualization mechanism to replace threads. submitted to PLDI 2011.

[19] Sean Halle and Albert Cohen. Dku pattern for performance portable parallel software, 2009. http://www.soe.ucsc.edu/share/technical-reports/2009/ucsc-soe-09-06.pdf.

[20] Sean Halle and Albert Cohen. Leveraging semantics attached to function calls to isolate applications from hardware. In *HOTPAR '10: USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[21] Carl Hewitt. Actor model of computation, 2010. http://arxiv.org/abs/1008.1459.

[22] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[23] G Leavens and M Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[24] Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.

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

[26] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *IEEE Computer*, 38:39 – 47, 2005.

[27] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 327–343, 2005.

[28] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 15–26, 1997.