For Extreme Parallelism, Your OS Is Sooooo Last-Millennium

ROB KNAUERHASE, ROMAIN CLEDAT, AND JUSTIN TELLER

Rob Knauerhase is a Research Scientist with Intel Labs. His current focus is observation-driven dynamic adaptation (resource mapping and scheduling) in system software. Other research interests include distributed systems, machine virtualization, and programming-system technologies for parallel applications. Knauerhase received a Master of Computer Science from the University of Illinois at Urbana-Champaign, and a Bachelor of Science in Engineering from Case Western Reserve University. He has 32 patented inventions, along with various and sundry publications and invited speaking events. Knauerhase is a Senior Member of the IEEE.
rob.knauerhase@intel.com

Romain Cledat is a Research Scientist at Intel Labs, where he has been contributing to Intel's research on exascale computing, particularly in the area of compilers and runtimes. Romain was previously a student at the Georgia Institute of Technology, where he received an MS in Electrical and Computer Engineering in 2005 and a PhD in Computer Science in 2011.
romain.e.cledat@intel.com

Justin Teller earned a BS degree in Electrical Engineering from Ohio University in 2002, an MS degree in Electrical Engineering from the University of Maryland, College Park in 2004, and a PhD in Electrical Engineering from Ohio State University in 2008. After graduating, Teller worked at Intel on high-throughput computing products before moving to Intel Labs to research scaling system software to exascale (and beyond).
justin.teller@gmail.com

Introduction

High-performance computing has been on an inexorable march from gigascale to tera- and petascale, with many researchers now actively contemplating exascale (10^{18}, or a *million trillion* operations per second) systems. This progression is being accelerated by the rapid increase in multi- and many-core processors, which allow even greater opportunities for parallelism. Such densities, though, give rise to a new cohort of challenges, such as containing system software overhead, dealing with large numbers of schedulable entities, and maintaining energy efficiency.

We are studying software and processor-architectural features that will allow us to achieve these goals. We believe that exascale operation will require significant out of the box thinking, specifically in terms of the role of operating systems and system software. In this article, we describe some of our research into how these goals can be achieved.

Motivation

Historic parallelism has come from banding processors together on a task, either in a large distributed system (whether in a grid, cloud, or other HPC/supercomputer configurations), a smaller cluster of server nodes, or a number of sockets on a motherboard. In each case, there are interesting problems for division of labor, interconnect tradeoffs (e.g., latency, bandwidth), and so forth. Concurrently, the microprocessor industry is trending toward many-core processors, uniting an increasing number of CPUs inside one processor; indeed, the recently announced Intel® Xeon Phi™ coprocessor [11] boasts more than 50 cores on a chip.

Meanwhile, processor fabrication technology has enabled cores to run at ever-lower voltages, which has worked out well for some high core count uses (throughput computing, graphics, etc.); however, even including optimistic public estimates of energy-efficiency improvements over time, one can extrapolate that an exascale

computer would require at least 500 MW of power, or roughly the output of a small nuclear power plant.

Despite the advances in many-core processor capabilities, we anticipate that exascale operation will require a staggering amount of computational resources. Such extreme systems will also entail an unprecedented amount of complexity in managing the effective coordination and use of resources. In our research, we have been exploring how notions of system software—current and old ideas, along with some new ideas—should change in order to run an exascale machine effectively.

Philosophy

In 2010, the Defense Advanced Research Projects Agency (DARPA) generated a solicitation for innovative joint-research proposals [3] on extreme-scale systems. The DARPA challenges, in addition to performance, included aggressive energy efficiency (both in terms of performance/watt and minimizing energy spent on intra-system communication), dynamic adaptability (to changes in workload, hardware, or external goals), and programmability. Our philosophies, software prototypes, and experiments have been strongly influenced by participation in this program.

Our research hypothesis for exascale system software is that *a fine-grained, event-driven execution model with sophisticated observation and adaptation capabilities* will be key to exascale system software. To this end, we break applications down into dataflow-inspired [4] codelets [18], which are invoked by a runtime environment based on satisfaction of data and control dependencies. Dependencies are specified either by the programmer or by a high-level compilation system.

Extreme-Scale Hardware

Based on current trends and our own hardware research efforts, we anticipate that a number of hardware characteristics will be typical of exascale systems.

As mentioned, in contrast to the previous upward spiral of core speed, industrial trends have been producing designs that include a larger number of modest-speed processors. Although the optimal arrangement of these cores—number per die, die per chip or socket, and so forth—is far from clear, our prototypes assume that hardware will (must) provide an efficient—fast, low-latency, low-power—inter-core communication mechanism.

Further, we expect that power requirements will require cores that are relatively simple, perhaps with shared-ISA heterogeneity to allow different alternatives for hardware acceleration. In the same vein, we anticipate that extreme-scale systems are all but certain to operate cores at near threshold voltage—with implicit challenges for reliability—while also allowing for dynamic voltage/frequency scaling controlled by system software.

We predict that the nature of I/O within the system will change. As fabrication processes and architectures never-endingly drive computation to be less expensive, communication will become relatively more expensive. At exascale, communication (moving a bit) is likely to be at least as expensive as computation (manipulating that bit), which requires a new focus on efficient data placement and code/data positioning optimizations from software.

Lastly, we foresee changes in memory organization. Power concerns will dictate a larger amount of per-core memory, perhaps scratchpad memory, additional register files, or various types of cache. Also, there will almost certainly be similar structures at a block (grouping of cores) level, and/or at a chip level, and so forth. Given the propensity of hardware architects to use fractal designs, replicating interconnected structures, our software and hardware prototypes comprehend a deep memory hierarchy, with classes of latency/power memories: local, group-local, neighbor-group-local, and so forth. Such structures provide both challenges and opportunities for system software.

Unsuitability of OS Functions for Exascale

Traditionally, the role of the operating system is to provide a variety of services in order to expose a common programming interface to programmers and applications, irrespective of the underlying hardware. Thus, an OS *actively* tries to abstract away and hide the nature of the hardware. This approach has proven to be quite successful, enabling the programmer to focus on the essence of his application without worrying about the nitty-gritty of managing hardware devices, memory, or threads and tasks. Also, this approach enabled programmers to write once and run their code unmodified on different generations of processors; however, our research has led us to believe that many of the traditional OS functions are in the best case suboptimal, and in the worst case directly counter to the energy and performance goals we are seeking. To that end, we have become iconoclastic about the notion of a traditional operating system for extreme-scale systems and have focused on either omitting (avoiding the need for) or simplifying much of the functionality one imagines in an OS. Instead, our research is exploring the development of a sophisticated, yet lighter-weight, *runtime environment* to manage an exascale machine.

Extreme-Scale System Software

In this section, we introduce the programming model we envision for exascale systems. In the following sections, we will specifically delve into memory and thread management.

Programming Model

Our programming model is heavily inspired by the dataflow-style codelet ideas we recently described in [18]. Dating as far back as the late 1980s [4], dataflow programming and machines are not new, but we re-examine these ideas in a contemporary environment so that we can benefit from both the insights and the shortcomings of the prior work.

The model decomposes traditional tasks into stateless codelets; after a codelet finishes, both the code and context can be destroyed safely, thereby conserving energy. It includes an explicit description of the dependencies among codelets and between codelets and data to allow for better co-location within an extreme-scale system. Codelets may additionally be automatically derived from higher-level representations, such as Concurrent Collections [12].

A key objective of the programming model is to reduce energy consumption by facilitating the co-location of data and computation, thereby devoting a larger part of the energy available to actual computation (as opposed to merely shuffling data around). Furthermore, explicit data and computation placement will allow high-

level tools, such as compilers and performance analysis toolkits, to provide the runtime with greater insight into the performance and energy usage of a program, thereby enabling the programmer to home in on "energy bottlenecks."

LEVERAGING META-INFORMATION

Our research is also investigating means to pass more information from the programmer and compiler into the runtime. In a traditional compilation system, data available to the compiler (or generated by the compiler's analyses) is not made available to the OS, being largely discarded when the binary is created. One example is the tradeoff between code space and loop overhead when a loop is unrolled: under different conditions, different versions of the code will have higher efficiency. Our research indicates that preserving awareness of tradeoffs like this can enable a runtime to more intelligently (co-)locate codelets, increasing performance and saving energy. The compiler can also generate various versions of the code, allowing the runtime to choose, based on environmental conditions and system goals, which version to run.

Other metadata will also prove crucial in a heterogeneous system as the codelets are able to expose their requirements or preferences in terms of hardware. The runtime scheduler then appropriately schedules these codelets, trying to optimize for their preferences, available resources, environmental constraints (such as temperature in various parts of the chip), and data locality.

Runtime Environment

Programs written to the model above are executed by a lightweight (relative to traditional operating system kernels) runtime environment such as our Intel Research Runtime (a component of the nascent Open Community Runtime [17]), or commercial alternatives such as the SWARM runtime [15, 5]. The runtime design deliberately empowers "hero" programmers by exposing more of the hardware features and execution details while still allowing "regular" programmers to write correct code.

The runtime leverages the fine-grained nature of codelets to allow explicit placement of code: for example, allowing a group of codelets which communicate or use the same data structures to be topologically grouped together for energy efficiency. Absent such direction, it can automatically use tuning hints or observation to (re-)locate code throughout the system. In particular, in a heterogeneous environment, it can select among alternate codelet implementations based on availability and proximity of certain cores. Likewise, for data placement, the runtime enables direct access to memory features such as DMA and inter-core networks, and can autonomously move data closer to code using techniques described below.

Because our anticipated hardware allows very fine-grained power and clock control of cores (clock-gating or changing frequency to save energy, and turning off unused hardware), the runtime also manages the overall energy profile of the system. Given different external policies (e.g., "deliver answer as fast as possible regardless of energy used" versus "take time to deliver an answer with minimal power consumption") and overall system state (e.g., voltage-related or permanent core failures), the runtime will turn on or off various parts of the system and schedule codelets in accordance with the specified policy. For example, if the policy specifies that the end-goal is performance, the runtime would schedule everything as fast as it could on the most powerful cores it could find. On the other

hand, if energy was a concern, the runtime may turn off certain cores and therefore utilize fewer resources.

SEPARATION OF CONTROL

An important function of the OS is to manage access from user code to hardware resources. Traditionally, a kernel runs in a privileged execution mode while user code runs with fewer privileges and has specific channels (system calls) to interact with the hardware. Implicit in most OS implementations is the need to switch between user and kernel modes. This switch incurs overhead which in some cases can be very expensive (hundreds of cycles) [16]. For complex exascale machines, such overhead (in terms of time, as well as power) is likely to be unsupportable.

Our system software similarly separates *control code* and *execution code* but does so in a way that enables us to get rid of many expensive OS functionalities. The runtime designates a small number of cores to be "control engines" (CEs), which execute the runtime itself in a distributed fashion across the system, and the majority to be "execution engines" (XEs) that run user code.

Our system thus disregards the notion of ring boundaries entirely, choosing instead to separate privileges by space rather than by time. Application (user) code is only run on XEs, and our control software runs on CEs. This division obviates the need for traditional processor "modes" and their associated overhead. Several types of security concerns are likewise alleviated; combined with hardware features (e.g., configurable range-checked access control and "locking" to render portions of memories immutable) and associated compiler support, our system obtains much of the security benefit of user/kernel division. Our simulations include hardware support to implement fast and power-efficient communication between the XEs and their controlling CE to replace the functionality traditionally provided by system calls.

This division of duties also allows specialization of each core type for power and performance as desired. For example, CEs do not need accelerator hardware for advanced math, but they may benefit from special low-latency interconnects or particular atomic instructions specialized for queue processing. Likewise, depending on the assumed needs of applications, XEs can contain variably sized floating-point hardware, optimized implementations of arithmetic and transcendental functions, or other hardware (e.g., encryption logic) as needed.

ABSENCE OF DEVICE DRIVERS

Since only system code runs on the CEs, there is no need for device drivers in the usual sense of the term. If a CE does not directly connect to a peripheral, it can forward its request to a CE that does. The system runtime for CEs that do directly support devices includes device functions (e.g., to manage device state, or to accommodate special instructions or interfaces) as appropriate.

A drawback of this approach is the difficulty in supporting a wide array of peripherals. However, in the time frame of our research, exascale machines are unlikely to be off-the-shelf commodity designs, and their owners are likely to have programmers or contracts to support hardware upgrades. User-code requests that entail peripheral devices are treated as data dependencies by the scheduler and are satisfied through runtime code on the CEs.

Memory Management

The OS has traditionally abstracted hardware in its memory management, in particular through the use of virtual memory, which provides, in part, a contiguous address space independent of the underlying hardware.

Modern OSes rely on virtual memory to provide each individual process with its own address space, thereby hiding the details of the backing physical storage medium (whether it is RAM or disk) from the programmer and allowing him to ignore the issue of code overlay. Virtual memory, however, also comes with two major costs: (1) the hardware and energy costs of doing the translation between virtual and physical addresses and (2) the loss of visibility into the varying characteristics of the physical memory (distance, energy costs, etc.).

The former can be dealt with [6] and is not a focus here; however, the latter is exacerbated in exascale systems where physical constraints make it necessary to have deep memory hierarchies to be able to simultaneously provide fast memory for data actively used by computations as well as very large ones for input and output data. The loss of visibility into the memory hierarchy can have drastic energy consequences. For example, without considering the energy required for the memory controller, it takes about 75 picoJoules per bit to move data from DRAM while it only takes about 0.5 pJ per bit to move data within the chip. This double-order of magnitude difference means that applications and their data must be very carefully positioned so that energy may be spent on computation rather than communication.

Can Virtual Memory Be Improved to Perform with Exascale?

Virtual memory provides undeniable advantages and, before dismissing it, we should consider whether it can be improved to provide better visibility and energy efficiency in exascale systems.

Fundamentally, the important issue is reducing the *distance* between data and computation. At first glance, virtual memory could be modified to provide this benefit as it adds a level of indirection (between the virtual address and the physical one). One can imagine a system where this mapping is influenced by the closeness of the physical page to the computation. This mapping could, and would, dynamically change at runtime as usage of the page changes. For example, suppose there are two cores, C1 and C2, each with its own RAM (M1 and M2). While threads on C1 are accessing a particular virtual page, it could be mapped to M1 where it is as close as possible to C1. However, if threads on C1 stop accessing it and threads on C2 start accessing it, the virtual page could be remapped to a physical page on M2 to minimize access costs.

APPLICATION FOCUS VERSUS SOFTWARE FOCUS

The problem with the above solution is that virtual memory is managed in an application-agnostic manner at a granularity that only makes sense from a hardware perspective. In particular, a single page may contain objects that have very different access patterns and would ideally be placed in two different physical locations. A programmer could force these objects to live in different pages, but this could result in huge memory fragmentation.

Solution: Make Data Objects First Class Citizens

The solution we are pursuing is to do away with all the memory "support" mechanisms an OS provides and allow the programmer to directly manipulate *data-blocks* as first-class entities. Note that HPC programmers are already bypassing OS memory support by writing custom allocators that are tuned to a particular architecture and paging mechanism. We posit that taking this further is essential for exascale computing.

Conceptually, a data-block is simply a contiguous chunk of memory with metadata annotations to allow it to be moved throughout the memory hierarchy. At a high level, data-blocks are similar to pages but bear a crucial difference: they are *semantically meaningful*. In other words, a data-block only contains data that is related in a way that makes sense to a particular computation. This is not true for pages (especially "huge" pages such as those supported in Linux), which can contain data pertinent to different and unrelated computations.

Note that data-blocks also introduce a level of indirection (since their base address in memory can change), but this can be efficiently dealt with in hardware and with compiler support. In spite of this additional indirection, the advantages provided by using data-blocks in exascale systems are very appealing:

- Each data object can be precisely placed in memory.
- As access patterns change, the objects can be moved either by the programmer, or automatically by the runtime. This also enables multiple cores to pass the same data object to each other (as in a pipeline) and have it optimally placed at each stage of the pipeline.
- Runtime observation systems, with the assistance of hardware, can track accesses to data objects, thereby enabling optimizations based on access patterns.

Note that the increased control given to the programmer does not necessarily mean that she has to manually manage everything and deal with all the complexities of fine-grained memory management. We envision a system where a runtime, aided by programmer hints or sophisticated observation via hardware performance-monitoring units, would optimally move data-blocks so as to reduce the energy required to access data.

Scheduling Management

The threading metaphor is perhaps the only parallel abstraction a modern OS exposes to the programmer. However, the scheduling granularity of threads is ill-suited for new programming models (namely, our own, as well as others such as Cilk, TBB, Habanero, X10, Chapel and Fortress).

Thread Parallelism for Exascale

A traditional OS is responsible for mapping threads to hardware resources and context switching them as needed to ensure that they all get equal access to computing resources. While this approach eliminates the need to know the number of underlying parallel resources—in line with the OS's objective to abstract away the hardware—emerging parallel programming models rely on being able to *precisely* schedule much smaller-grained tasks and frequently implement a runtime layer on top of traditional threads to manage dependencies and the mapping of tasks to OS-provided threads.

These fine-grained schedulers and the OS may have misaligned goals (the OS being more concerned about fairness, for example, which may be less of a concern to certain runtimes that may favor critical path execution), and this leads to kludges in the runtime to ensure that the OS does not perform "optimizations" that harm performance. These hacks include (1) using processor affinity to prevent thread migration, and (2) carefully matching the number of OS threads to the number of hardware-supported threads to avoid context switching among others. The latter hack demonstrates that the runtime is actively attempting to break through the OS abstraction and trying to get to the underlying hardware characteristic (here, the number of hardware threads). Essentially, the runtime and the programmer work very hard to avoid many of the services the OS provides since the goals of the OS (fairness in execution, responsiveness, etc.) are not shared by exascale programming.

CONTEXT SWITCHING

Specifically, avoiding context switching is paramount in exascale systems since the cost of context switching in terms of energy (as well as time) is nontrivial. Furthermore, to reduce the energy needed to access data, extreme-scale systems will increase the amount of hardware-provided thread-local storage such as registers and private scratchpad memories. In this situation, avoiding context switching becomes all the more critical.

Our program decomposition into small codelets [18] allows us to run a single codelet per XE without overhead from preserving ephemeral state (swapping registers, managing scratchpads, etc.). Since XEs are plentiful, the scheduling system can narrow or widen its scheduling of parallel code according to system state, workload, and overall power/performance tradeoff policies. Additionally, as the runtime understands that a finishing task's context is no longer needed, it can put that core into a very low-energy (destructive to memory) sleep state.

Required Threading Notions

The scheduling of codelets really only requires two notions: affinity and dependencies.

AFFINITY

A modern OS will provide hooks for threads with the affinity interface expressing links between threads and hardware resources. Conversely, our runtime provides an interface to define affinity between and among tasks. This approach allows the runtime to better understand the relationships among all tasks, such as which tasks are related and therefore share data. With this information, the runtime can

schedule tasks onto the hardware resources in such a way to increase locality, or to optimize for other system operation goals. Furthermore, since the entire programming system and runtime are aware of the presence of possibly specialized XE hardware, the programmer is able to create specialized tasks to run on that specialized hardware. Both the compiler and programmer are also able to generate multiple equivalent versions of a task to run more efficiently on a heterogeneous computing substrate.

DEPENDENCIES

Dependency information is also entirely expressed within the runtime's tasking interface. This concept is not novel among runtime interfaces (the TBB task graph [10] is one example). However, our runtime makes this dependency information available to the lowest levels of the system software. One particularly exciting area we are exploring is how this interfaces with the memory management portion of the runtime. As the memory subsystem can see how tasks are related and vice versa, the runtime can make better scheduling decisions. For instance, tasks can be scheduled to minimize data movement by either relocating data or moving a task to execute closer to its data. As all the dependencies are known to the runtime, related tasks can be moved at the same time, leading to an overall more efficient system. This data-directed scheduling research is an area from which we anticipate significant results from our work and that of the broader community-related work. Many of the ideas expressed above are admittedly not entirely our invention, with some having a rich heritage of prior research. Previously, however, these ideas were developed separately, independent both of an extreme-scale (hardware, software, and energy) focus and of tradeoffs among interesting combinations. As process technology and new architectures continue to drive many-core systems to unprecedented levels of parallelism, we are reaching a stage where bringing all these ideas together in a cohesive system and testing them on real hardware will not only help validate them, but also provide concrete solutions to the power-efficiency problems facing the industry.

Other participants in DARPA's UHPC program [2] have identified similar concerns for performance and energy efficiency along the lines of what we describe here. Current HPC technologies, such as MPI [7, 9] and OpenMP, reflect a large software investment and will also need to be improved and scaled to exascale machines; what we propose in this article does not preclude these other developments. Efforts such as Kitten [14] by Sandia National Labs, IBM's Blue Gene CNK [8], and the Barrelfish research OS [1] also aim to support extreme-scale systems with varying focus on performance, power, and scalability.

Conclusion and Future Work

We believe that extreme-scale systems are likely to break many of the known and beloved design conventions of both hardware and software. Existing challenges for performance, power efficiency, adaptability, and programmability will be greatly exacerbated when going to exascale operation. Our research, therefore, is pursuing some fairly radical concepts with respect to traditional definitions of OS functionality. We are taking advantage of hardware and software co-design to experiment with separating control and application duties across heterogeneous cores, as well as resurrecting prior dataflow-inspired techniques in our codelet execution model.

While much work remains to be done, results to date are encouraging. Our simulations show good reduction in the overhead (especially energy) that a traditional OS would incur, especially in terms of context-switching and memory management. Leveraging prior work, we plan to introduce more dynamic observation features in our runtime and hope to further optimize both power and performance by automatically migrating data closer to code and/or code closer to data.

Exascale computing has become a primary interest throughout industry, academia, and government. In partnership with both academia and government, we are continuing to explore ideas such as those discussed in this article. With continued progress, we hope to usher in a new HPC era, in which scientists and engineers will have access to a new generation of "extreme-scale" supercomputing systems with unprecedented computational power *along with* supportable energy consumption.

Acknowledgments

We are indebted to Shekhar Borkar for championing our research in the exascale community. Thanks also to DARPA and the Department of Energy for their interest in (and funding of) exascale HPC research. We are also grateful to the HotPar'12 workshop community for their feedback on the paper on which this article is based [13], and to USENIX ;login: for giving us the opportunity to share our research in this forum.

This research was, in part, funded by the US Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Government.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture For Scalable Multicore Systems," *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (SOSP), 'ACM, 2009, pp. 29–44.

[2] B. Dally, "Power, Programmability, and Granularity: The Challenges of Exascale Computing" (IPDPS keynote): http://techtalks.tv/talks/54110/, 2011.

[3] DARPA, UHPC BAA: http://tinyurl.com/38zvgm4, March 2010.

[4] J.B. Dennis and G.R. Gao, "An Efficient Pipelined Dataflow Processor Architecture," *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, 'IEEE Computer Society Press, 1988, pp. 368–373.

[5] Eti Web site: http://www.etinternational.com/.

[6] D. Fan, Z. Tang, H. Huang, and G. Gao, "An Energy Efficient TLB Design Methodology," *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED '05), 2005,* pp. 351–356.

[7] A. Geist, "MPI Must Evolve or Die," *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 2008, p. 5.

[8] M. Giampapa, T. Gooding, T. Inglett, and R.W. Wisniewski, "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK," *Proceedings of the 2010 ACM/IEEE International Conference for High Performance*

Computing, Networking, Storage and Analysis (SC '10) IEEE Computer Society, 2010, pp. 1–10.

[9] W. Gropp, "MPI at Exascale: Challenges for Data Structures and Algorithms," *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 2009, p. 3.

[10] Intel threading building blocks: http://www.threadingbuildingblocks.org.

[11] Intel Xeon Phi coprocessor: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[12] Intel Concurrent Collections for C++ 0.7 for Windows and Linux: http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/, 2011.

[13] R. Knauerhase, R. Cledat, and J. Teller, "For Extreme Parallelism, Your OS Is Sooooo Last-Millennium," *HotPar 2012: 4th USENIX Workshop on Hot Topics in Parallelism,* USENIX, 2012.

[14] Sandia National Laboratories, "Kitten Lightweight Kernel": https://software.sandia.gov/trac/kitten, 2012.

[15] C. Lauderdale and R. Khan, "Towards a Codelet-Based Runtime for Exascale Computing: Position Paper," *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (EXADAPT '12) ACM, 2012, pp. 21–26.

[16] J. Liedtke, "On Microkernel Construction," *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15),* ACM, 1995.

[17] Open Community Runtime Google code repository: http://code.google.com/p/opencommunityruntime/, 2011.

[18] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G.R. Gao, "Using a 'Codelet' Program Execution Model for Exascale Machines: Position Paper," *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (EXADAPT '11), ACM, 2011, pp. 64–69.