Operating Systems for Exascale

John Shalf, Thomas Sterling, Costin Iancu, Katherine Yelick, John Kubiatowicz

Introduction

Chip Multiprocessors containing hundreds or even thousands of cores will challenge current OS practices. Many of the fundamental assumptions that underlie current OS technology are based on design assumptions that are no longer valid for a Chip Multiprocessor (CMP) containing thousands of cores. In the context of exascale system requirements, as machines grow in scale and complexity, techniques to make the most effective use of network, memory, and processor resources will also become increasingly important. The interaction of the OS with the application is closely intertwined with the execution model for parallel applications where parallel tasks interact.

Thomas Sterling's "four horsemen" analogy for the problems that will afflict exascale computing (Latency, Starvation, Contention, and Overhead) can be used to understand the impact of the OS on the execution model. Execution models that support more asynchrony will be necessary to hide *latency*. Such execution models will also require more carefully coordinated scheduling to balance resource utilization and minimize work *starvation* or resource *contention*. These execution models will also require extraordinarily low *overhead* fine-grained messaging. However, the attributes required by the execution model are nearly impossible to achieve when the OS intervenes for every operation that touches its privileged domain – it must intervene for inter-processor communication operations, has exclusive and privileged control of scheduling policy, and exclusive ownership of resource management policies. A baseline challenge for the exascale software stack is how to get the OS out of the way without compromising the need to protect hardware state from errant (or malicious) software. Otherwise, it is highly unlikely that novel programming models can emerge that are capable of addressing the emerging constraints of extreme-scale systems.

Hardware Evolution Direction

We make the following assumptions about future system architecture as we move towards exascale systems.

- Heterogeneous cores with potentially different ISAs (old systems would have just one ISA)
- Hundreds to thousands of cores per chip (old systems would have just one)
- Billions of cores on system (this is a radical departure from the past)
- Less memory per core (after years of continued growth)
- Not fully-connected topology between cores (even for SMP's a PRAM model has been assumed)
- Component failures on order of seconds or minutes (formerly system-scale checkpoint-restart to disk was sufficient)
- Fault recovery creates load imbalances
- Power is the leading design limiter, which puts more emphasis on power management capabilities (power was not formerly a top-level issue)

What is wrong with current Operating Systems

Hardware is rapidly evolving away from the original design assumptions of modern operating systems. However, they have evolved to be far too large to fix using an incremental path. Over time, Operating Systems have evolved into multifaceted and hugely complex software implementations that have accreted a broad range of capabilities. We refer to the challenge of

breaking the OS apart based on separation of concerns as "deconstructing the OS." Below are a subset of leading issues that motivate the need to reexamine the underlying assumptions that are encoded in current OS implementations and fodder to imaging a re-implementation of the minimum necessary capabilities based around emerging realities of hardware design.

Time Sharing

Time-sharing OS's are built around the assumption that the CPU is a precious resource that must be shared. This is no longer true when you have a CMP with thousands of cores and constrained memory bandwidth.

Old Conventional Wisdom (CW): When CPUs are considered the most precious resource, you time-multiplex access to share access. However, when hundreds of CPUs are available, it no longer makes sense to suffer the overheads incurred by context switching. Indeed, the cost of a context switch is not merely the time spent preserving registers because the associated cache pollution (and other shared resources) can substantially reduce CPU throughput. Context switching makes inefficient use of the **new** precious resources, which are on-chip memory and off-chip bandwidth.

New CW: If cores are cheap, then you spatial partition the allocation of cores for function rather than offering timeslices of a single resource. The spatial partitioning is analogous to Logical Partitioning (LPAR) from 1970's era mainframe terminology.

Research in this area: MITOSYS (MIT/Berkeley), K42

Device Drivers and Resource Sharing (Quality of Service)

OS's play an important role in virtualizing finite hardware device interfaces – making it appear to each application that it is exclusive owner of a replicated copy of the device interface. In the parallel context current OS's assume workload of uncoordinated processes that stochastically vie for control of finite/virtualized devices.

Old CW: OS's currently implement a greedy allocation policy where first process acquires a lock to gain exclusive access device (or OS/driver interface) for each I/O transaction. This approach is sensible for stochastic access to the virtualized resource, but very bad for highly-synchronous parallel algorithms. Resource and lock contention hurts performance by flooding the inter-processor communication network with redundant/spinning lock acquisition requests. Locks also serialize otherwise parallel processes when they attempt to access the same resource (the network interface for example), and subjects them to nondeterministic delays.

New CW: A new OS will need use a QoS management for symmetric device access by a large number of entities or put coordinated scheduling of device access in control of the application. The development of novel mechanisms for coordinating parallel access to finite number of virtualized device interfaces is essential. (e.g. Ncpus >1000 and the Ndevice=1).

Examples of Research: NOOKs, K42

Interrupts and Asynchronous I/O

Background task handling for asynchronous operations are currently handled by threads and signals in modern OS's and application designs. In particular, devices must interrupt the CPU in order to invoke the device driver software to service their requests.

Old CW: Interrupts and threads (a side-effect of time-multiplexing access to a core) are used to implement asynchronous operations or system calls in user codes. Even more fundamentally, device handling is implemented by interrupting the CPU for the time-critical top-half of the device driver and then scheduling the bottom half of the driver to run on the next available time-slice of the CPU. The interrupt subjects processes to non-deterministic delays that can result in load-imbalances for parallel applications.

New CW: If CPUs are abundant, side-cores can be dedicated to asynchronous I/O and device handling. For that matter, cores can be used to implement programmable DMA and async I/O rather than using dedicated hardware components for said operations. Using a core or

background thread to implement DMA eliminates the need to write to the device control interface for dedicated DMA engine, which require costly msync() operations by the requesting CPU.

Exampels of Research: Side-Cores

Fault Isolation

Another important role of operating systems on more robust systems is fault isolation. This is particularly difficult for large SMP systems as errors can propagate rapidly through the system and are extremely difficult to track down. With growing node concurrency, and increasing likelihood of soft errors, the ability to rapidly identify and isolate errors will be essential. The total lack of a fault isolation strategy in modern OS's is arguably one of their weakest points moving forward.

Old Conventional Wisdom (CW): CPU failure on a chip multiprocessor (or SMP) will result in a Kernel Panic that takes down the system. Such events will happen with increasing frequency in future silicon.

New CW: CPU failure should result in isolation of the partition containing the failure. It would be better yet if the hardware supported integrated rollback mechanisms to support partition Restart. There is existing work in the Singularity OS to consider how transactions between the application and device interfaces can be rolled-back to a known state to support restart of partitions containing the application.

Research Examples: VMM containers, Singularity

Inter-Processor Communication

All device interfaces are at the OS's privilege level in a typical system. Therefore, any access to virtualized device interfaces is mediated by the OS in order to protect the hardware interface from misbehaving (or malicious) programs. However, the overhead of the privilege change and additional data buffer copies required for OS mediation has given way to a wide variety of "OS Bypass" and "user space messaging" implementations such as VIA, PORTALS, and DRI. However, these approaches expose the hardware to irrecoverable state corruption by errant software. Whereas current OS works in-band to protect the hardware state, it would be more efficient to employ extra cores or some other supervisory hardware to monitor application-to-device transactions out-of-band to check for state corruption without adding additional overhead to the communication stream.

Old CW: invoked for ANY interprocessor communication or scheduling to protect hardware state from corruption.

New CW: direct HW access allowed by application, but hypervisor monitors for state corruption out-of-band from the transactions (perhaps using dedicated subset of cores as observers).

Research Examples: Singularity, MVIA/MVAPICH (very old example that is isolated to OS bypass for MPI)

Serialized Interrupt Dispatch

Old CW: vectors interrupts to a handler code that starts at a given memory address.

New CW: vector should include both the handler code **AND** the ID of the CPU core that should vector to the interrupt address. This will be even more critical for heterogeneous multicore systems. Moreover, current interrupt controllers tend to serialize the interrupt stream. We have not yet conceived of a hardware mechanism for dispatching event interrupts concurrently on a CMP, nor have we thought of how to implement a software infrastructure to manage such a concurrent event stream.

Examples: none to date.

There are numerous other issues with current operating system designs that this paper will not cover in detail. These include:

Global Resource Discovery and Global Addressing: Conventional TLB isn't designed with global memory in mind. If we think global memory is important, then this problem must be addressed.

Memory Hierarchy: Most operating systems depend entirely on conventional cache-coherent memory hierarchy, while many new hardware platforms are evolving toward unconventional (not necessarily cache-coherent) memory hierarchies

Heterogeneity: Operating systems generally depend target single-ISA homogeneous systems. An accelerated architecture will t

Power Management: Fine grained Power Management control is not directly accessible to user-space applications

Locality Management: The default policy for most operating systems is opportunistic allocation of resources, which can be overridden with explicitly set process affinity, which will not scale to larger more complicated chip multiprocessors.

Redesigning the OS to Meet NEW Requirements

An incremental approach to modifying Linux to address the divergence of hardware design constraints from fundamental OS design assumptions is unlikely to make sufficient progress in the 10 year timeframe we have for exascale systems. OS's must be refactored (deconstructed) to offer more flexible resource management and runtime support for parallel execution models with the focus on exposing system resource usage policies to the various level of the programming stack. The overall goal of a deconstructed OS would be to allow the application to compose the best resource usage policies for its particular needs and to adapt to system scale and load. Policy control should be hierarchical, with different levels of abstraction depending on their consumer. For example, a hypothetical communication scheduling mechanisms exposes to the libraries/compilers explicit control over message sizes and ordering, while exposing to the application level/programmer only abstract policies like "long routes first". Adaptation can take form of Quality of Service mechanisms or migration for communication locality.

Previous experience with compiler and runtime optimizations for parallel applications indicates that lightweight control over OS mechanisms is not sufficient for good performance and additional control over the policies that guide the management of these mechanisms is required. Looking beyond existing languages and execution models, even more control of resource management will be necessary to support the kind of novel execution strategies required for Exascale applications. Current OS designs favor generic policies, e.g. preemptive thread scheduling or Least Recently Used page replacement, which have been selected as being the least common denominator for commercial workloads. The execution models required for structured parallel algorithms in the scientific computing applications are less diverse, usually require cooperation between computing entities and exhibit a relatively ordered execution imposed by data/task dependence. When developing performance-critical parallel applications developers are willing to trade time to market for more effort to fine-tune the performance of their application. Compilers for parallel languages are also well suited to take advantage of such functionality. The challenge is of course to offer some level of control over resource usage policies while still meeting the productivity goals of the development environment.

Exascale hardware technology is also envisioned to be highly memory constrained. Therefore, rather than a full OS model, there may be substantial benefit from an Exokernel model where applications are bundled with only the necessary OS functions linked in to the application that run in their own partition, relying on a partition manager for defining protection, resource sharing, and management of Quality of Service (QoS) guarantees without interposing itself on every resource

request. We will refer to these protection mechanisms as a “partition” because there are a number of technologies including hypervisors, VMMs, and runtime environments such as Singularity that can implement this kind of isolation in a spatially partitioned CMP. The primary roles of the application container are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the application container will have complete control over scheduling and virtualization, if any. For example, in an SPMD execution layer used in UPC, there is no need for processor virtualization, while for dynamic threading used in the DARPA HPCS languages, a lightweight user-space thread scheduler that can be directly controlled by the application or runtime would be beneficial.

Minimalism/Modularity: A very thin protection layer should resident on the CMP to prevent hardware state corruption, but otherwise offer bare-metal access to the underlying hardware wherever possible. Many system facilities will be linked at user level as a set of optional systems libraries. Hardware protection mechanisms will allow direct, user-level access to facilities such as networking and I/O through the lightweight protected messaging layer. Parallel applications will be given bare metal partitions of processors that are dedicated to the given application for sufficiently long to provide performance predictability. The primary roles of the the protection layer are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the protection layer will have complete control over scheduling and virtualization, if any. There are many lessons to be learned from K42 [1], the MIT Exokernel [2], and embedded operating systems such as VxWorks [3] regarding approaches to efficient and modular system services. The thin protection layer (possibly a hypervisor or VMM-based application container) will play a role in mediating concurrent access to devices to ensure fair share of resources. Although software functions can be virtualized through replication, hardware devices are finite and access to them by multiple hardware components must be managed. This will include Quality Of Service guarantees for access to certain rationed resources such as memory or network bandwidth.

This approach to modularity was also employed by microkernels. The implementation of channel-based modularity was the Achilles heel of microkernels because you had to cross protection boundaries for any inter-module communication, which incurred a substantial cost due to the context-switch overhead. There are a number of demonstrations by the K42, Tesselation, and Plan-9 operating systems that indicate that the overheads for such an approach are very well controlled right now.

Isolation: Groups of processors can be combined into protected partitions. Boundaries will be enforced through hardware mechanisms restricting, among other things cross-partition shared memory. Messaging between partitions can be restricted based on a flexible, tag-based access control mechanism. OS functionality such as device drivers and file systems will be both spatially distributed and than time-multiplexed; we refer to this as spatial partitioning. This approach works in synergy with the sidecore techniques [4], which allow an application to vector OS or driver functions to free cores rather than forcing a context-switch on the core running the application.

Safe User-Level Messaging *(not cache-coherent shared memory)*: Messages will be used to cross protection domains rather than more traditional trap-based mechanisms. Through hardware mechanism and/or static analysis, applications will have direct, user-level access to network interfaces and DMA. Further, fast exception handling and hardware dispatch of active message handlers will permit low overhead communication without polling. Most traditional system-calls will translate into messages to remote cores (other system-calls will be to linked system libraries) [4].

Related Work

Operating system and programming language development is a work intensive process with a long initial development time. Traditional operating systems have a monolithic design with little or no control over the internals exposed to the application or user level. Novel programming models need to demonstrate clear performance and productivity advantages over the established paradigms in order to have a chance for widespread adoption. Their efficiency can be greatly improved when having access to fine application level control over functionality provided by the system software stack (OS). The same fine-level of control is beneficial to established execution models of existing parallel runtime environments.

There are several DOE sponsored ongoing research projects related to operating systems design. The ZeptoOS [5] project distinguishes between service node and compute node kernels and provides tool for OS instrumentation and understanding of the interaction between the OS and the application layer. The Plan-9/Right-Weight Kernels [6] project focuses on a judicious selection of OS level services in order to diminish OS interference. The K42 research project focuses in parallelizing the OS itself and providing abstractions for overall system adaptation at scale. A common characteristic of these projects is the focus on improving overall system behavior and reducing OS interference [7] and putting more resources under the application and user space control while maintaining fault isolation and security. The mostly explore the opportunities offered by lightweight kernels and focus on kernel level mechanisms for resource control: CPU, memory, and network.

Hypervisors, VMM-based application containers, and various code-rewriting systems offer a very thin protection layer that promises to offer the following capabilities for massively parallel CMP-based systems. One approach to both operating systems and runtimes for parallel execution is to deconstruct conventional functionality into primitive mechanisms that software can compose to meet application needs. A traditional OS is designed to multiplex a large number of sequential jobs onto a small number of processors, with virtual machine monitors (VMMs) adding another layer of virtualization. One alternative approach is to explore the usage of a very thin partition management layer that exports spatial hardware partitions to application-level software. The partition management layer sets up protection to prevent interaction between in order to isolate errors and faults on chip. The partition manager is responsible for setting up communication between partitions that enable safe inter-partition communication and interaction. Lastly, the partition manager can grant far more control to the runtime system to support advanced programming models, while maintaining security.

This approach allows the program within each partition to use custom processor schedulers without fighting fixed policies in OS/VMM stacks. The hypervisor supports hierarchical partitioning, with mechanisms to allow parent partitions to swap child partitions to memory, and partitions can communicate either through protected shared memory or messaging. Traditional OS functions are provided as unprivileged libraries or in separate partitions.

For example, device drivers run in separate partitions for robustness, and to isolate parallel program performance from I/O service events. An alternative “deconstructed” architecture would enable partitioning not only of cores and on-chip/off-chip memory but also of the communication bandwidth between these components, with QoS guarantees for the system interconnect. The resulting performance predictability improves parallel program performance, simplifies code (auto)tuning and dynamic load balancing, and supports real-time applications.

Finally, the Berkeley ROS and Tesselation projects represent a ground-up rewrite of current operating systems with the manycore design point in mind. These operating systems partition resources in both time and space. ROS, in particular creates an abstraction for a manycore process, which is absent in current mainstream OS implementations, which gives control over

scheduling policy to the code that runs within the manycore process (a two-level OS). Tesselation provides similar capabilities, but packages this management solution as Cells, which can package multiple address spaces to support embedded multi-level OS's. The design goals for both OS implementations are to provide finer grained control over resources to provide better QoS guarantees and more control over resource management to applications and runtime systems.

Risks

Most OS research and development efforts tend to be an all-or-nothing proposition, with software deliverables reaching the large user community very late in the project life cycle. It has therefore been very difficult to bring radical new OS ideas into mainstream usage – making this kind of research a very high-risk activity.

Biblography

[1] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. IBM Systems Journal, 44(2):427–440, 2005.

[2] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[3] Wind river home. www.windriver.com.

[4] S. Kumar, H. Ra j, K. Schwan, and I. Ganev. The Sidecore Approach to Efficient Virtualization in Multicore Systems. In Submitted to HotOS 2007, 2007.

[5] ZeptoOS: The Small Linux for Big Computers. Available at http://www-unix.mcs.anl.gov/zeptoos.

[6] R. G. Minnich, M. J. Sottile, S.-E. Choi, E. Hendriks, and J. McKie. Right-Weight Kernels: an Off-the-Shelf Alternative to Custom Light-Weight Kernels. SIGOPS Operating Systems Review, 40(2), 2006.

[7] F. Petrini, D. Kerbyson, and S. Pakin. The Case of Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC03), 2003.