A Minimum Overhead Operating System Kernel for Better Scalability in HPC*

Daniel Rubio Bonilla[1], Colin W. Glass[1], Lutz Schubert[2]

[1]HLRS - Universität Stuttgart, Germany
[2]Universität Ulm, Germany

rubio@hlrs.de, glass@hlrs.de, lutz.schubert@uni-ulm.de

Abstract. In this paper we propose a new design of an Operating System Kernel especially designed for HPC (High Performance Computing) systems. As technology is reaching the limits of silicon physics, it is not possible to substantially improve the CPU core's performance by simply increasing the frequency or through new internal structures as there are no new ideas on how to increase single threaded performance substantially. This had led to a situation in which the theoretical performance of HPC systems can only be increased by integrating more CPU cores, making it necessary for the applications to be able to instantiate more and more threads to be able to make use of the increasingly amount of resources in the system. But often the overhead and jitter of the Operating System can make the application to perform worse as the level of parallelization increases. Currently, most HPC systems use adapted general purpose Operating System kernels in the compute nodes. In this paper we will discuss what problems this carries and we will describe an especially designed kernel that addresses HPC needs arguing that it can greatly help in reducing the Operating System's overhead and contribute to improve jitter, enabling higher levels of concurrency and improving overall application performance by allowing more heavily threaded applications.

Keywords

Operating Systems, Kernel, HPC, overhead, jitter

1 Introduction

For decades now, the computing industry has relied on increasing clock frequency and architectural changes to improve performance. The situation has changed nowadays; small architectural improvements continue, but the power wall, the memory wall and thermal issues have made the approach of increasing clock frequency unfeasible. Semiconductor manufacturers continue developing the silicon manufacturing technology and doubling the number of transistor per unit area every few months, but rather than to improving the clock speed (because the physics do not allow it any more), they increase the number of processing units (i.e. cores) in the chip die.

This trend has been an uptake by HPC (High Performance Computing), continuously multiplying the number of nodes and core count in the HPC machines as the main way to increase the (theoretical peak) performance. Creating a situation in which only embarrassing parallel applications can effectively increase the execution performance by taking advantage of the additional resources. To illustrate this trend to, for example, 20 years ago, the most powerful computer had 140 cores (Numerical Wind Tunnel), 10 years ago 51200 (BlueGene/L) and in November 2013 over 3 million cores (Thianhe-2) [14].

In order to maximize the theoretical performance of new HPC system, developers must use an extremely high level of parallelism in their programs, rather than expect higher performance from the processor updates. But exploiting parallelism in any degree can be difficult: not only that the data must be segmented, but the communication overhead can lead to reduced performance. Also, the amount of work of concurrent threads must be sufficient to compensate for the overhead of thread creation and management. Unrolling loops with

* Work funded by BMBF (Bundesministerium für Bildung und Forschung) grant 01IH13003 - MyThos Projekt

short or little compute-intensive loop bodies can be too expensive and can greatly reduce the performance of the program.

The adequate load per loop body or per thread is dependent on several factors, the first is the code and data size transmitted to the destination node. It also has to be taken into consideration the operations executed by the Operating System to instantiate and configure the thread and the additional services needed to make the execution of the threads possible. Although in some Operating System kernels implementations threads have fewer dependencies than processes, the minimum execution environment for threads is still too large and leads to cache misses, instantiation challenges and expensive context switches when a thread is rescheduled or makes a system call.

This overhead is not just a problem that the programmer should cater for, but it is also a primary obstacle that keeps parallel applications from scaling better because it prevents the effective use of concurrency. Several studies have noted that the current architecture of Operating Systems generally have a negative impact on scalability and performance, and that an Operating System redesign for HPC is urgently needed [1][8][6][2].

Current approaches to increase scalability try to improve the data segmentation, to produce better adapted thread bodies and to reduce data dependencies. However, the Operating System is part of the core of the problem, and has not yet been correctly addressed. To correctly address this issue the architecture of Operating Systems for highly scalable applications has to be redesigned to eliminate secondary functionalities and reduce as much as possible the overhead.

Most computers in the Top500 use monolithic general purpose kernels, such as Linux, that internally have feature inter-dependencies of features that are not relevant for the execution of individual threads, but that increase memory and CPU usage [14]. For example, due to the expensive system calls, currently the creation of a thread requires over 10,000 compute cycles [8][16]. Although thread pool can be used to reduce this time, it also reduces the degree of dynamicity of the system.

Our aim is to reorganize the Operating System architecture so that it is exclusively dedicated to tasks related to computing, specially multithreading, with some extra functionality necessary for creating the execution environment specially designed for our purpose, the execution of HPC applications. These non-core functionalities of the Operating System are loaded accordingly to the needs of the system or the application at each moment. With this reorganization we expect to greatly increase the performance for thread creation.

To achieve this a modular approach will be used, taking concepts from the microkernel architecture paradigm, which means that the resources will only be minimally loaded by the Operating System, leaving more free resources to the program itself as in each node there will be deployed the minimal functionality for it to work. But this deployment will be done by generating a single execution context for the application and the Operating System, reducing the Operating System overhead, by having less expensive system calls, and improving the predictability of the system, which ultimately increases the performance of distributed synchronous systems.

This approach proposes an Operating System implementation dedicated to rapid deployment and management of threads on specific hardware architectures. The modular design of the Operating System will allow programs to interchange data between threads instantiated on different system architectures, supporting communication across different system boundaries (i.e. processor core to processor nodes, etc.). We expect that our Operating System kernel design to be a suitable solution for increasing the scalability of applications in multicore processors and multiprocessor nodes.

2 Current Operating Systems in HPC

Modern conventional Operating Systems are designed to support a wide range of needs and for this they provide a large set of interdependent daemons. In the computing nodes there might also happen to be hardware that is not needed by the program that is running at that moment and that is causing hardware interruptions. These processes (daemons) and interruptions that is not relevant to attend lead to a high overhead, unpredictable amount of CPU and memory usage and context switches.

The unpredictability of the Operating System and the system daemons behaviour affect specially communications. Current HPC systems typically use Message Passing Interface (MPI) or similar mechanisms for communication, as well as synchronization across computing nodes. Developers introduce specific synchronization events in the application code but if one core takes longer to reach to that point then all the other processors must wait, increasing the overall execution time as all nodes execute at the speed of the slowest. One of the main reasons that makes a processor take longer to reach a synchronization point than the others is the unpredictable Operating System and daemons behaviour making HPC systems particularly sensitive to the Operating System footprint and overhead.

Many HPC vendors use custom Lightweight Kernel Operating System (LWK) that try to reduce this overhead by pursuing these goals and characteristics:

- Target massively parallel environments composed of thousands of processors with distributed memory with a tightly coupled network.
- Provide support for highly scalable, performance-oriented scientific applications.
- Emphasize efficiency over functionality.
- Maximize the amount of resources (e.g. CPU, memory, and network bandwidth) allocated to the application.
- All previous characteristics have the common aim of “Minimize time to completion for the application” [4].

The implementations of LWK may vary but they all try to give applications predictable and maximum access to the CPU, memory and other system resources. They often do this by simplifying algorithms such as thread scheduling and memory management and reduce system daemons to a minimum as computing nodes do not have to cater for multiple purpose computer services, such as enabling a multiuser and multitasking enviroment. In LWK systems, available services, such as job launch, are constructed in a hierarchical fashion to ensure scalability to thousands of nodes. Networking protocols for communication between nodes in the system are also carefully selected and implemented to ensure scalability, reducing latency (by reducing the network stack) and maximizing bandwidth [4].

By restricting services to only those that are absolutely necessary and by streamlining those that are provided, the jitter of the LWK Operating Systems is reduced, allowing a significant and predictable amount of the processor cycles to be given to the parallel application. Since the application can make consistent forward progress on each processor they will reach their synchronization points with better synchronization, as result, the node’s waiting time is reduced [12].

While computing nodes run the application and execute the calculations, the service nodes take care of the extra functionality. These are usually a small set of nodes running full service-like Operating System to offload services from the computing nodes (login access, compilation, job submission/launching, file I/O, etc.). All these functionality is needed for allowing users to run their application and interact with the system while providing security for the system and the user’s data.

Some good examples of systems that use custom LWK Operating Systems in the computing nodes are:

- Compute Node Linux (CNL): is a runtime environment based on the Linux kernel for the Cray XT3, Cray XT4, Cray XT5, Cray XT6, Cray XE6 and Cray XK6 supercomputer systems based on SUSE Linux Enterprise Server. It is part of the Cray Linux Environment (CLE) [18].
- The IBM Blue Gene supercomputers run various versions of Compute Node Kernel (CNK) Operating System based on Linux [7].
- Sandia National Laboratories has an almost two-decade commitment to Lightweight Kernels on its high-end HPC systems. It includes Operating Systems such as Catamount that was deployed on ASCI Red, and continues its work in LWKs with the Kitten OS [11].

Most of current LWKs are based on Unix-like Operating Systems, mainly using Linux, which is a monolithic general purpose kernel. In chapter 2.3 we will analyze that many of the general purpose Operating System kernel features are not needed in HPC computing nodes and can affect negatively the performance of the applications.

2.1 HPC System Architecture

In current HPC systems the activities of the computing nodes are orchestrated by “clustering middleware”, a software layer that sits on top of the nodes and allows the users to treat the cluster as one large cohesive computing unit (via a single system image concept).

The service nodes are the single point of management and job scheduling for the HPC, providing control and access to the computing resources and queuing the user’s jobs (programs they want to run) and their associated tasks; allocates resources to these jobs, initializes the tasks on the computing nodes and reports the status of jobs, tasks, and compute nodes if necessary.

The most typical way of using this kind of systems consist in the user accessing the service node, sending their application source code and data, then compiling the application source code into a binary using an optimized compiler designed for the specifics of the system and its architecture and linking it to an also a set of optimized mathematical and communication libraries (such as BLAS or MPI). Then the user commits the compiled application with its data to the execution queue, that will decide when and on which specific computing nodes to execute the application. Once the execution is finished the user can retrieve the processed data from the system and run the application again with new data if desired.

In this paper, when we refer to a HPC Operating System, we are exclusively referring to the Operating System that is running in the computing nodes and not to the service or I/O nodes. The computing nodes are in most computers exclusively running one application of one user at a given time (although different nodes may execute different applications simultaneously), in other words, computing nodes are monouser and monotask in their nature.

The computing nodes work in an isolated environment, in which they do not have direct access to the "external world", and the only communication point that they have is through the service nodes. That means that the computing nodes cannot suffer direct attacks from the external world. The advantage of this design is that there is no need to introduce security mechanisms in the computing nodes as all the possible attacks have to go through the service nodes, which should provide the convenient security mechanisms. The service node handle the user requests and determine the amount of computer nodes that each application and each user has available at each moment.

2.2 Current Issues

Even if they reduce the Operating System jitter, current node LWK Operating Systems are still far from an ideal situation as usually they still have these problems [15]:

- Context Switching overhead: Changing from the CPU's user mode to kernel mode is in current systems very expensive. It has been measured, on the basic request "getpid", to cost 1000-1500 cycles on most machines [9]. Of these just around 100 are for the actual switch (70 from user to kernel space, and 40 back), the rest is kernel overhead. In modern microkernels, such as the L3 microkernel, the minimization of this overhead reduced the overall cost to around 150 cycles [5]. But for other complex tasks typical microkernels are slower; microkernel Operating Systems attempt to minimize the amount of code running in privileged mode, for purposes of security and elegance, but ultimately sacrificing performance.
  - No real need for CPU execution modes: the purpose of distinct operating modes for the CPU is to provide hardware protection against accidental or deliberate corruption of the system environment (and corresponding breaches of system security) by software. Only trusted portions of system software are allowed to execute in the unrestricted environment (also known as "kernel mode"). All other software executes in one or more user modes. If a processor generates a fault or exception condition in a user mode, in most cases system stability is unaffected; if a processor generates a fault or exception condition in kernel mode, most Operating Systems will halt the system with an unrecoverable error. When a hierarchy of execution modes exists (ring-base security) faults and exceptions at one privilege level may destabilize only the higher-numbered privilege levels. Thus, a fault in ring 0 (the kernel mode with the highest privilege) will crash the entire system, but a fault in ring 2 will only affect rings 3 and beyond and ring 2 itself, at most.

    This feature is not strictly needed in HPC Computing Nodes as they only hold code and data of the application that is being executed in that right moment, so in case a user wanted to execute malicious code it could only attack itself. While at the same time it would not reduce functionality as faults and exceptions can still be handled in the most privileged mode.
- Cache Misses: Operating System kernel is too big, even when configured correctly the Operating System Kernel (normally Linux, 96.4% by November 2013 according to Top500 [10]) uses a notably large amount of memory. The impact on RAM usage is not noticeable (very few MiB vs many GiB), but the impact on processor cache is considerable. General purpose kernel's data structures are very big, as they have to contain extra information for features not needed in computing nodes, and are not designed to make correct usage of cache lines as they are not specifically adapted for the processor on which they are running and compilers are not perfect in transforming them, the data structures, when generating the binary code. Also the data and code in memory should be organized so that they take advantages of the processor's

data and instructions prefetch algorithms. The main idea is that the kernel should use as little as possible of the processor cache to evict the minimum of data / code belonging to the application [17].

- Operating System jitter: these systems still have many daemons running. In this kind of systems most of the daemons needed for a general purpose system has been removed, but some are still running. They provoke context switches, increasing overhead and evicting program's data and code from the processor's cache. They also reduce the predictability of the system worsening the inter-node synchronization times.

By improving the cache miss rate, eliminating context switches and improving the system's predictability we can expect applications to create smaller body threads, increase resource usage and improve scalability, leading to a better performance and smaller time to completion.

2.3 General purpose Operating System features that HPC does not need

As explained in chapter 2.1, HPC systems are by nature, at least at computing node level, single user, and single task systems. For this reason there is no need for the Operating System to offer multiuser and multitasking services provided by general purpose Operating Systems, making, in principle, possible to get rid of many of the general purpose Operating System's functionality:

- daemons: we should try reduce even further the amount of daemons running in the system (the ones left are monitoring, distributed workload management...) and if possible eliminate them completely. For this we need to find another solution to provide those services without using the standard daemon design. A solution is proposed in Chapter 3.
- big monolithic kernel: reduce even more the kernel size to minimize the disruption of the cache, and use simpler and faster system calls. The design is based on a minimalistic flexible microkernel that loads on demand the needed functionality by the application running on a specific computing node [13].
- context switches: context switches are expensive and introduce a lot of overhead. Try to reduce them or minimize the impact of them, or even try to completely eliminate them as proposed in Chapter 3.
- loaded drivers for all devices: to reduce memory usage and cache disruption, devices not needed by the application should be disabled. This also means that the system should disable interruptions sent from the devices not needed, which would also reduce context switches. It should be taken into consideration that some hardware, even if not used, will need some initial setup (for example, to enable power-saving modes). This can be done by loading the driver, executing the set-up, and then unloading the driver from memory, leaving the hardware correctly initialized while reducing the usage of the system's memory.

3 Single-Context Operating System and Application Environment

The minimal and modular Operating System kernel design proposed in this paper tries to create a tight coupling of the Operating System, hardware drivers, service daemons and the application in a single context, thus completely removing the context switches and creating a low overhead and more predictable execution environment.

The Figure 1 depicts how the Operating System and applications are mapped to memory in a general purpose Operating System. The kernel and the applications are mapped into the same virtual address space (usually the Operating System is mapped to the same addresses for every application virtual address space, for example Linux maps the Operating System to the higher half leaving the lower half of the address space for the application) but as the Operating System kernel and the application are running on different CPU rings, executing functions implemented in the Operating System from the application require the execution of a special CPU instruction that provokes a switch in the CPU ring and jump into the Operating System code. We have to be aware that usually daemons run as if they were just another application running in "user mode" CPU ring, implying that to schedule them on a CPU core a full context switch is needed.

Each kernel thread or application thread shares all the virtual address space of the kernel and the application except for the stack of each thread (for both kernel and application threads), that is mapped privately; the stacks are situated on the same address segment, but in different virtual address spaces so each thread has only access to its own stack. This system provides independent address space contexts for each thread, providing security between applications, as different programs cannot access data and code from each other, and also easier multitasking, as all applications can be mapped exactly to the same memory addresses without interference.

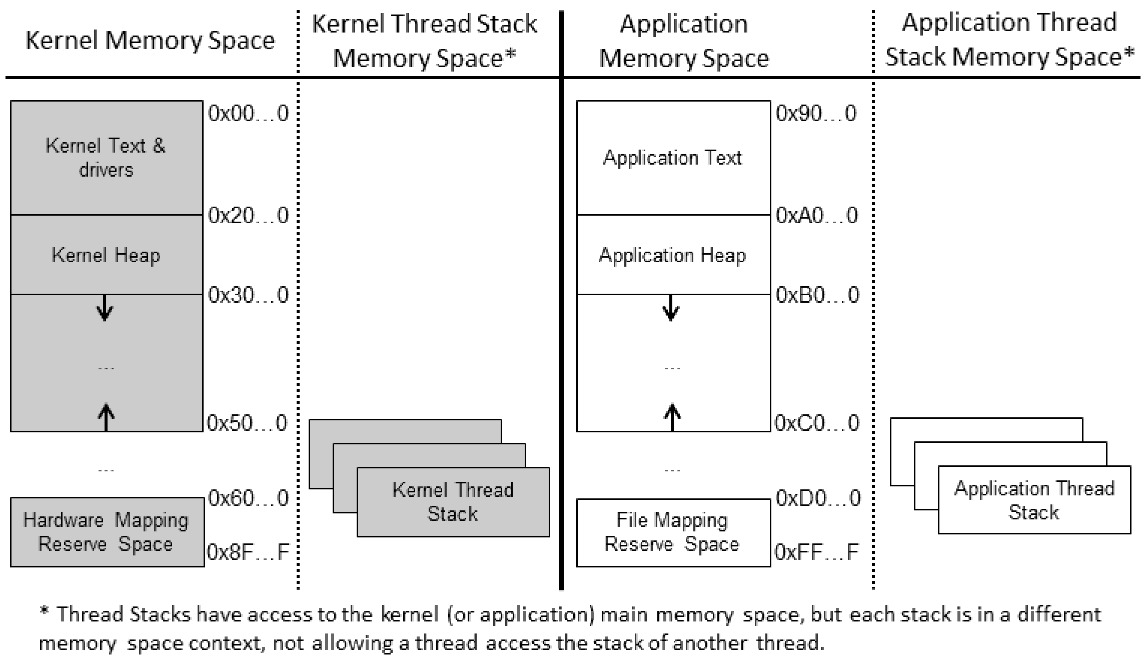
Kernel Memory Space
Kernel Thread Stack Memory Space*
Application Memory Space
Application Thread Stack Memory Space*
Kernel Text & drivers
Kernel Heap
Hardware Mapping Reserve Space
0x00...0
0x20...0
0x30...0
0x50...0
0x60...0
0x8F...F
Kernel Thread Stack
Application Text
Application Heap
File Mapping Reserve Space
0x90...0
0xA0...0
0xB0...0
0xC0...0
0xD0...0
0xFF...F
Application Thread Stack
* Thread Stacks have access to the kernel (or application) main memory space, but each stack is in a different memory space context, not allowing a thread access the stack of another thread.

Figure 1. Typical Operating System Memory Space organization, showing a main monolithic kernel image with a few kernel threads, and one application with many threads (there could be many applications executing, each with its own virtual memory address space). The kernel and the application are mapped to the same address space, except for the stack of each thread, but run on different CPU rings.

In our design we remove the multiple virtual address space contexts and merge them into a single address space, at the cost of decreasing inter-application security and, in principle, also multitasking facilities. This implies that we have to be able to map the Operating System kernel, service daemons, and application to the same address space, meaning that it has to be clear in the design the address range for each segment or we will have to use PIC (Position-Independent Code) in those systems that allow it. Figure 2 shows one alternative, of the many possible, of the memory organizations for a joined address space of the Operating System and the application. The segments we have to reserve in memory are these:

- Kernel Text: the binary code of the basic kernel and where the functionality extension modules are loaded.
- Drivers Text: hardware driver binaries.
- Daemons Text: binary code of the substitute of the functionality of the service daemons.
- Application Text: binary code of the user's application running on the computing node.
- A general growing heap: dynamically created data structures used by the kernel, drivers, daemons and applications are stored here.
- Thread stack space: individual stack space is reserved for each thread, with its maximum size fixed at the thread's creation point. The number of stack spaces will be the same as of threads instantiated in the computing node.
- Hardware and file mapping space: memory address space reserved for mapping opened files from I/O devices and hardware resources.
- Firmware mapping space: space reserved for mapping the computing node's firmware (a.k.a. Bios or UEFI in x86 based systems).

The memory address layout shown in the Figure 2 is an example of a possible organization the main "memory blocks", and the final organization will depend on the particularities of the different hardware architectures. Also the ranges of addresses shown for each memory block are examples and have not been determined yet as this requires further study and will also change depending on the underlying hardware architecture.

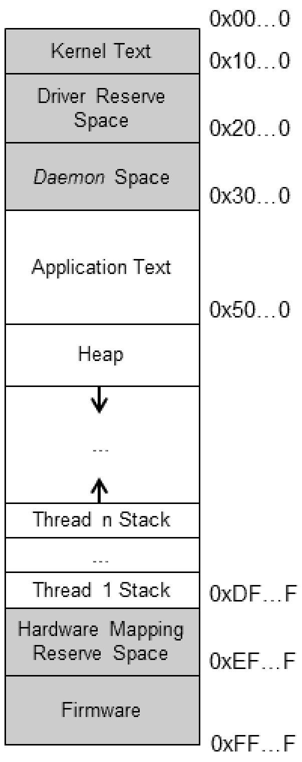

Figure 2. An example of a new single context memory organization

In modern Operating Systems each thread stack is in a different memory address space (even when they share the rest of the address space with the other threads of the same application), but this would imply that changing the execution thread on a certain CPU core would imply to set-up the new stack, that is forcing a context switch, affecting negatively the Operating System overhead (because a context switch, on most architectures, flushes the whole TLB, thereby forcing page table walks, which implies a few memory reads). In our case, for changing the execution thread on a CPU core would be enough with setting correctly the stack registers.

From the user's point of view, with this new design the application concept and behavior will remain the same as in current systems with the service nodes providing the user services (login, compilation, security...), there will be no changes on how the system is used or the programming model for the applications; users can continue using the system in the same way and they can program their applications using the same programming language, except for some of the Operating System calls.

The complexity of the service nodes would not be increased with the model discussed in this paper as most of the modifications would have to be implemented into computing node, the toolchain and the deployment of the application.

4 Application Life Cycle and Toolchain Modifications

To be able to successfully achieve the generation of single context tasks that are embedding the application, the Operating System, hardware drivers and the service daemons a new compilation schema is needed. Depending on how the toolchain and software stack is finally designed and implemented there might be the drawback that the application has to be recompiled, or at least relinked, whenever there is an update in the system's hardware or main parts of the software stack (such as when the kernel's ABI is changed). Although this situation could be mitigated by using PIC code and dynamically linked object code (similar to Windows DLLs or ELF shared libraries) in other situations.

The compiler has to be aware of the location of the Operating System calls and services, has to integrate the interruption handling with the application, and be aware of where the application text is going to be placed in memory. This design allows that many typical small Operating System calls could in principle be inlined in the application making these operations much faster than in typical Operating System kernel but the situations in which this possibility is used might be carefully analyzed as the code duplication could in some architectures increase the cache miss rate, especially in often used functionality.

Conventional Operating System designs the system calls are implemented by the application throwing a trap, a.k.a. a software interruption (INT in x86 architecture [3]) a parameter that is the identifier of the desired system call, usually stored in a register. This is necessary in order to raise the CPU ring and decreasing it again to return from the system call. In our design, as the application is merged with the Operating System, executing under the same CPU ring and sharing the memory context, the system calls are implemented as simple function calls, thus greatly reducing the context switch overhead.

Figure 3 shows the application life cycle of the new design comparing it to current HPC application life cycle and pointing out the main differences:

0. Application design and implementation: the user writes the application and debugs it in its own system, as the redesign of the Operating System kernel affects the computing node software stack and modifies some of the common Unix/POSIX system calls a special library should be provided so that the application can run on top of the Operating System of the developer's workstation (this can easily be achieved by the library translating the system calls into the native ones).
1. Access to the HPC machine: the user logs into the service node where he stores the code of the application and the data, as done in current systems.
2. Compilation: then the application is compiled using the specialized toolchain. This step is the same for the user as compared to current system but as described before, the binary generated by the compiler will have a different structure.
3. Application submission to the execution queue: the user submits the application and the data to run to the HPC workload management, which will automatically determine the assignment of the computing nodes and execute it in the moment it considers appropriate.
4. Computing Node setup: link the minimal Operating System with the necessary modules, drivers and the daemons replacement code with the application, generating the single context task binary for each different type of node in case of an heterogeneous environment, otherwise a single image might be enough.
5. Deployed on the computing nodes: deploy the binaries generated in the previous step accordingly with the requested topology for the execution of the application and the resources assigned by the workload manager of the service nodes.
6. Execution: run the application and store the generated output in the service nodes (or the I/O nodes), where the user can retrieve de results).
7. Cleaning: the computing nodes are "cleaned"; the application is removed and computing node is ready to accept new applications.
8. Service nodes can launch a new application as determined by the workload manager.

We can conclude that the application life cycle suffers some changes from the point of view of how it is internally handled by the HPC system but from the user's point of view it remains mostly unchanged.

5 Possible limitations of a Single Context Operating System

The new kernel design proposed in this paper imposes a set of limitations in its capabilities and functionalities in comparison to modern general purpose kernels, such as Linux, of which the most relevant are:

- Lost of Address Space Contexts: this happens because the application and the Operating System are running in the CPU's kernel ring and sharing the memory address space. As the computing nodes are isolated from the external world, software attacks do not have a posibility to happen. The security with the "external" world is controlled ty the service nodes, that includes I/O operations and user data control.
- No Multitasking: there will be no possibility of multitasking inside a computing node, that should not be confused with a single task being able to execute many threads, which is something needed. This should not change how HPC systems work, as currently they do not offer multitasking at Compute Node level, even if in principle it could be possible.

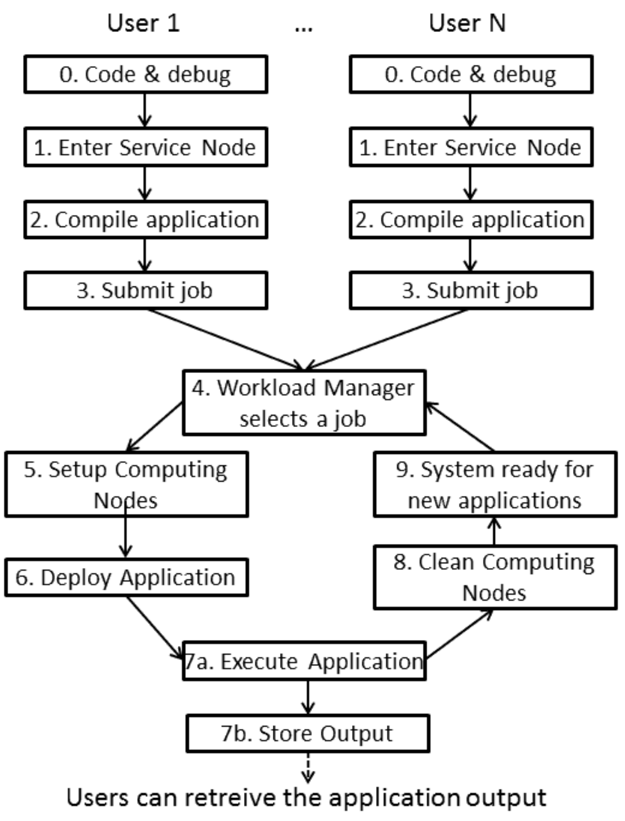

Figure 3. Application life cycle in HPC with the proposed Operating System design

- No Multiuser: to allow multiple users in the same system first you need multitasking. As this is not supported, then multiple users is also not supported. But as with the multitasking, this will not generate any impact in current HPC systems as they currently do not support multiple users at the compute node level.
- Crashed Application Recovery: a computing node might completely halt (crash) because of a hardware failure or a software bug. In convectional systems the Operating System can cater for crashes affecting applications by closing them and reseting the status of the computing node to a usable state to accept new applications. But even conventional HPC systems need mechanism for externally resetting a computing node in case the Operating System kernel crashes, for example to recover from a Linux "kernel panic". In the proposed design many of the application's bugs and the Operating System crashes will have to be resolved externally, this means we might need the computing node resetting mechanisms more often than in current systems, but these mechanisms are, in any case, already implemented.

In summary, it can be observed that we lose some features of modern general purpose Operating Systems, but these are the features that, even if available, are not used in the HPC computing nodes making our approach feasible without affecting the user experience and without the need of a different hardware design.

6 Conclusion

In this paper we have argued that general purpose Operating System kernels, even when customized, are not well suited for modern HPC computing nodes as these do not need kernel features that are currently integrated and just increase the unpredictability of the system (affecting negatively the Operating System jitter), worsen the cache miss ratio and have high overhead for system calls.

For this reason we propose a complete redesign of the Operating System kernel for HPC, taking into consideration the modularity concept of microkernels and loading in run-time the features needed by the compute nodes and the application with the aim to reduce system's overhead, improve the cache miss rate and increase the predictability. All this together will allow system calls to execute faster, and improve the synchronization of the system, enabling applications to be decomposed in smaller threads, as the amount of work of the body of the thread does not need to be so big to compensate the thread creating overhead, increasing the level of usage of resources of modern HPC systems due to the better scalability, thus reducing the total time to completion of the application.

This is achieved without big changes from the user's perspective as all the changes are integrated in the Operating System and the toolchain that generates the binary code of the application. As the proposed kernel design and functionality is a new philosophy and implies changes that drop not needed functionality provided by general purpose kernels, many of the typical system calls of Unix/POSIX systems will have to be modified with the implication that compatibility with legacy code that uses the eliminated or modified calls will have to be adapted. But it should be considered that many of these often provided calls by general purpose kernels are not often used in HPC programs or they will be provided in a different, and often more simple, manner.

As we progress the research, development and evaluation of this new design, further publications will be published addressing the challenges that emerge and evaluating different solution, detailing the algorithms and implementations used.

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'09*, p. 29–44. ACM, 2009.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. Frans Kaashoek, R. Morris, and N. Zeldovich: An analysis of Linux scalability to many cores. *Proceedings of the 9th USENIX conference on Operating Systems design and implementation, OSDI'10*, p. 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[3] Intel Corporation: Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*, version 048, retrieved on January 2014.

[4] S. Kelly, R. Brightwell: Software Architecture of the Light Weight Kernel, Catamount. *Proceedings of the 2005 Cray User Group Annual Technical Conference*, 2005.

[5] J. Liedtke: On micro kernel construction. *Proceedings 15th ACM Symposium on Operating System Principles*, SOSP, December 1995.

[6] Y. Ling, T. Mullen, and X. Lin: Analysis of Optimal Thread Pool Size. *ACM SIGOPS Operating Systems Review*, 4(2), p. 42-55. ACM, 2000.

[7] J. Moreira, et al: Designing a Highly-Scalable Operating System, The Blue Gene/L Story. *Proceedings of the 2006 ACM/IEEE International Conference for High-Performance Computing*, SC'06, 2006.

[8] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat: OctoPOS, A Parallel Operating System for Invasive Computing. *Proceedings of the Systems for Future Multicore Architectures workshop*, 2011.

[9] J. Ousterhout: Why aren't Operating Systems getting faster as fast as hardware? *Usenix Summer Conference*, Anaheim, p. 247-256, 1990.

[10] Top 500: Development over Time of Operating System Family. *http://top500.org/statistics/overtime/*, November 2013.

[11] R. Riese, R. Brightwell, P. Bridges, T. Hudson, A. Maccabe, P. Widener, K. Ferreira: Designing and Implementing Lightweight Kernels for Capability Computing. *Concurrency and Computation: Practice and Experience* 21:6 p. 793-817 April 2009.

[12] G. Di Sirio: Response Time and Jitter in CHIBIOS/RT. *CHIBIOS/RT Documentation*, October 2011.

[13] L. Schubert, A. Kipp, B. Koller, S. Wesner: Service Oriented Operating Systems. *IEEE Wireless Communications*, IEEE Wireless Communications 16(3), p. 42-50, 2009.

[14] E. Strohmaier: Top Spot on 42nd TOP500 List. *www.top500.org*, November 2013.

[15] A. Tanenbaum: Modern Operating Systems. *Upper Saddle River, NJ: Pearson/Prentice Hall*, p. 160. ISBN 978-0-13-600663-3, 2008.

[16] M.W. van Tol: A Characterization of the SPARC T3-4 System. *CoRR*, 2011.

[17] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, X. Ji: Adapting cache line size to application behavior. *Proceedings of the 13th international conference on Supercomputing*, p. 145-154 ICS'99, 1999.

[18] D. Wallace: Compute Node Linux, overview, progress to date, and roadmap. *Proceedings of the 2007 Cray User Group Annual Technical Conference*, 2007.