SAND2015-3988C

Panel: What is a Lightweight Kernel?

Rolf Riesen
rolf.riesen@intel.com

Arthur (Barney) Maccabe
maccabeab@ornl.gov

Balazs Gerofi
bgerofi@riken.jp

David N. Lombard
david.n.lombard@intel.com

John (Jack) Lange
jacklange@cs.pitt.edu

Kevin Pedretti
ktpedre@sandia.gov

Kurt Ferreira
kbferre@sandia.gov

Mike Lang
mlang@lanl.gov

Pardo Keppel
pardo.keppel@intel.com

Robert W. Wisniewski
robert.w.wisniewski@intel.com

Ron Brightwell
<rbbrigh@sandia.gov>

Todd Inglett
todd.a.inglett@intel.com

Yoonho Park
yoonho@us.ibm.com

Yutaka Ishikawa
yutaka.ishikawa@riken.jp

ABSTRACT

Lightweight kernels (LWK) have been in use on the compute nodes of supercomputers for decades. Although many high-end systems now run Linux, interest in options and alternatives has increased in the last couple of years. Future extreme-scale systems require rethinking of the operating system, and modern LWKs may well play a role in the final solution.

In the course of our research, it has become clear that no single definition for a lightweight kernel exists. This paper describes what we mean by the term and what makes LWKs different from other operating system kernels.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design

Keywords

High Performance Computing; Multi kernels; Hybrid kernels; Lightweight kernels

1. INTRODUCTION

Light Weight Kernels (LWKs) have been in use in supercomputers since at least the early 1990s. Today, most systems on the Top500 list [18] run Linux®, a Full-Weight Kernel (FWK). Figure 1 illustrates this dramatically. The graphs show what percentage of the top 500 systems run a given OS. Twenty years ago, in 1994, there was a rich diversity of OSes in use. Most vendors had their own OS to differentiate themselves from others, but also to fully support their hardware, which was not standardized at all.

Although the 2014 graph in Figure 1b is broken down into categories, almost all of them rely on the Linux kernel. The different categories represent different versions of the same Operating System (OS) and additions by various vendors. This means that users of these systems are familiar with Linux and expect similar behaviors and features even on non-Linux systems.

In the last couple of years interest in LWKs, and OS research in general for the very top-end of High Performance Computing (HPC), has increased again. This is due to anticipated architectural changes in future extreme-scale systems, as well as new directions in use and programming of these systems. Other reasons for the renewed interest are that FWK are slow to adapt to the increasing demands of top-end HPC, and concerns about scalability and fault tolerance at the very high end.

OS changes will be necessary. Maybe these changes will come in the form of enhancements to existing OSes or be more radical. Maybe bringing back LWKs from the past, updating them, and making them coexist with Linux, will be a successful path to an OS for the extreme-scale future.

mOS [29] is a path finding project at Intel[1] where we research options for such an OS. In the course of our work we realized that the term is not well defined. The purpose of this document is to explain what we mean by Light Weight Kernel (LWK), work with the community to refine the definition of the term LWK, and differentiate the principles of LWKs from similar technologies such as microkernels [16] and library operating systems such as Libra [6].

2. LWK CHARACTERISTICS

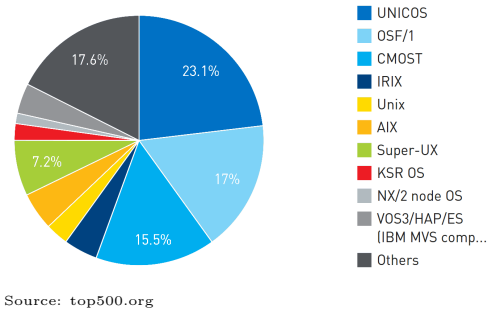
In this section we list the components of a traditional LWK, and then look at specific characteristics and attributes of LWKs. We close this section with a short discussion of microkernels.

2.1 Components of a traditional LWK

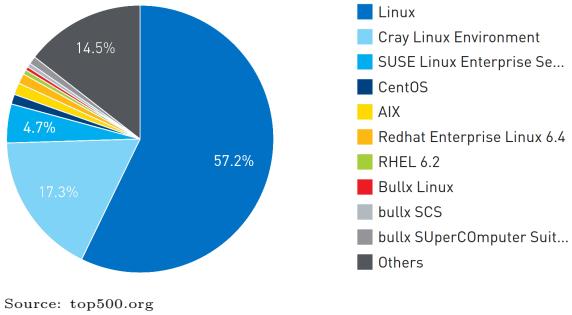
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

[1]Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

ROSS '15, June 16, 2015, Portland, Oregon, USA
ACM 978-1-4503-3606-2/15/06.
http://dx.doi.org/10.1145/2768405.2768414



(a) Usage in 1994



(b) Usage in 2014

Figure 1: OS usage change in Top500 systems over the last twenty years. OS names as provided by top500.org

Figure 2 shows the typical components found in a traditional LWK. IBM's®Compute Node Kernel (CNK) [11] is an example of such a kernel. Most of the blocks in that figure describe familiar OS components. Note that a virtual file system and local I/O are not strictly necessary. Some LWKs, such as SUNMOS [12, 17] and Puma [27, 25], had no notion of a file.

2.1.1 *Traps, exceptions, interrupts, and the system call interface*

In modern CPUs there are several ways of transitioning from user space into the kernel. Transitions are triggered by software, the processor, or by other hardware; and may be synchronous or asynchronous. Examples include synchronous memory exceptions, count-down timers, and interrupts from external devices.

We lump all of them together and respond to them in the box labeled "interrupt handling". Interrupt is meant in the generic sense: software or hardware interrupts, traps, or exceptions. We separate out those transitions that invoke system calls in the figure because they play such an important role in an OS kernel. The system call interface block is responsible for these.

2.1.2 *Process and memory management*

Process and memory management really define the behavior of an LWK. Most of the other components of an LWK are similar to the corresponding components in an FWK or are omitted. Process and memory management stand out because they are often much simpler and specialized in an LWK. They avoid buffering in the kernel and specialize using larger page sizes which are well-suited for extreme-scale applications. Low overhead and fast performance are important, but deterministic behavior and scalability are given equal weight.

2.1.3 *Signal handling*

Signal handling is also similar to an FWK, although some LWK do not handle the full set of signals available in a Linux system. Control and handling of those signals may also differ. Especially in earlier LWKs, many signals simply cause the application to abort or are ignored.

2.1.4 *Machine check handling*

The machine check handling block deals with faults the system encounters. A typical example is the handling of an uncorrectable error in data read from memory. In an FWK these kinds of errors usually result in an application abort or even a system reset.

Some applications might be able to survive such faults, if they are informed at which memory location the fault occurred and are allowed to continue running. An LWK is not strictly necessary for this. Since version 2.6.32, Linux can generate a `SIGBUS` signal for many of those cases and allows an application to install a handler for that signal.

However, an LWK may make it easier to experiment with new ideas like that. Since the overall structure of an LWK is less complex, it is also easier and quicker to experiment and do research with them [7, 8, 9, 26].

2.1.5 *The hardware abstraction layer*

Most LWKs do not have a full Hardware Abstraction Layer (HAL). Nevertheless, even an LWK needs some functions to deal with the specific hardware it is running on. Examples include memory barriers, atomic operations, use of privileged instructions, and access to control registers.

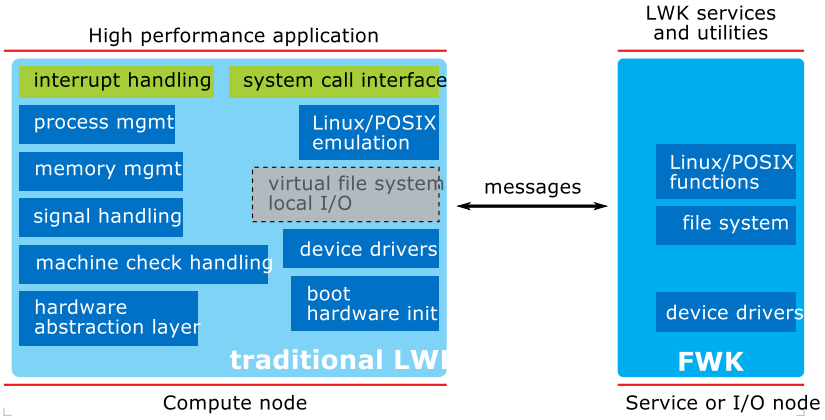


Figure 2: Components of a traditional LWK

These are encapsulated for ease of coding, not to make porting the LWK to other hardware easier. LWKs may be portable due to their small size and low complexity, but do not have a HAL to make them more portable than they already are. Direct access to new and unique hardware features is more important than portability

2.1.6 *Linux/POSIX emulation and virtual file system*

Because most LWKs attempt to be at least somewhat Linux and Portable Operating System Interface (POSIX) compatible, there must be functionality in the LWK for that. Some examples handled by that component are an Executable and Linkable Format (ELF) interpreter and loader to read and decode the image of an executable, and support for pseudo file systems; i.e., providing the contents of files such as `/proc/meminfo`. There is also a virtual file system layer for local I/O functions; e.g., **mmap()**, access to `/proc` and `/sys`, and maybe a RAM disk. This includes functions to open and read `/proc` files, if they are provided.

Of course, this is not the only POSIX compliant block in our diagram. We gave process and memory management, as well as signal handling, their own boxes.

2.1.7 *Device drivers*

There is usually a single device driver in an LWK: the one needed to enable and access the Host Fabric Interface (HFI). Occasionally there is code for a diagnostic device such as a serial line or a JTAG interface. But, there is no support for the hundreds of devices a Linux kernel can operate. Most of these devices do not exist or are not used in the systems that use an LWK. In a 10,000-node system, not every node has a mouse or speakers attached to it.

With more exotic and complex memory coming in the future, the LWK may need to provide a driver to take advantage of it. That doubles the number of devices an LWK has to deal with, but still leaves only a handful of devices for the LWK to manage.

On a compute node with less than a handful of devices, even an FWK would only have the few drivers built-in and activated which are needed. The difference to an LWK is that the LWK could not operate in an environment where more drivers would be needed.

On the other hand, the notion of a device may disappear entirely from an LWK. Similar to some LWKs that have no notion of a file. This greatly simplifies the LWK. No amount of configuration and specialization can achieve this kind of simplification with an FWK.

2.1.8 *Boot and hardware initialization*

Finally, just like any FWK, an LWK needs code to boot and initialize the hardware. Again, in some cases that may be simplified code because the LWK may not enable all hardware features present. For cost reasons it may be cheaper to use commercial, off-the-shelf motherboards to build a large-scale machine, even though some of the functionality provided by these boards will never be used.

There are a few things which are not shown as individual components in Figure 2. They include timer support and a diagnostic infrastructure which are commonly found in both LWKs and FWKs.

Also missing is an indication of a certain lack of functionality when it comes to hardware. For example, an LWK does not virtualize hardware (in the traditional sense), but it provides more direct access to the hardware. It is designed for OS bypass, while zero-copy/OS bypass devices are generally a special case in an FWK. LWKs also do not have page caches or buffering.

2.2 LWK attributes

There is no defining architecture of a "typical" LWK. The ones in existences differ significantly from each other, and several other types of OS kernels are also lightweight: nano, exo, micro, and embedded kernels come to mind. What we mean by an LWK is a kernel specifically designed and streamlined for the very high end of HPC; computers that deserve the name supercomputer, and a subset of applications and data sets that require machines like that. These systems have extreme scales and levels of parallelism.

In [23] we find this description:

> *"...lightweight kernels (LWK), which focus on providing exactly and only those services needed by specific hardware and a small set of mission-critical applications."*

This is in contrast to other *small* kernels, often called microkernels, that provide building blocks to create the desired services in user space. Supporting a limited application set makes the problem space smaller and allows for customized solutions. This does not mean LWKs cannot be composable

and adapt to different usage needs. It just means that this is a secondary goal; if it exists at all.

An LWKs main purpose is to make hardware resources available to scalable, parallel applications that can take advantage of it. The LWK maintains protection of the system and between users, if it supports multitasking or if the cores of a compute node are distributed among multiple users. It uses very few resources for itself and *gets out of the way*, so applications can use more CPU cycles and memory than what an FWK can provide. Furthermore, an LWK conditions the system for optimal use. For example, it reduces OS noise and provides large pages by default to minimize Translation Look-aside Buffer (TLB) misses. FWK attempt to do that too, of course, but they have to fight their own inherent complexity to do so. LWKs have a simplified virtual memory system and lack background activity, while tasklets and other myriad sources of soft interrupts make this difficult in Linux for example.

2.2.1 Design goals

LWK and FWK have very different design goals. The Wikipedia entry for *Lightweight Kernel Operating System* lists these design goals [28]:

- Targeted at massively parallel environments composed of thousands of processors with distributed memory and a tightly coupled network.
- Provide necessary support for scalable, performance-oriented scientific applications.
- Offer a suitable development environment for parallel applications and libraries.
- Emphasize efficiency over functionality.
- Maximize the amount of resources (e.g. CPU, memory, and network bandwidth) allocated to the application.
- Seek to minimize time to completion for the application.

Table 1 takes these goals and compares them to those of an FWK. By now it should be clear that LWK shoot for minimal functionality to provide exactly what is needed, but nothing more. By doing that, LWKs can achieve better performance and scalability.

Light weight is achieved by an LWK in several ways. Looking at Figure 2 we see that several components are missing which are usually present in an FWK: e.g., a file system and a network stack. But it is the extreme simplification of the remaining components that is the hallmark of an LWK. We look at several in turn.

2.2.2 Extreme simplification: Process management

Although an LWK has process and memory management components, they are smaller and less complex. For example, many LWK support a single task per CPU. Even if multiple tasks are allowed, they are running using cooperative, non-preemptive task scheduling. A lot less code is required for that than the very complex code and data structures required by Linux' Completely Fair Scheduler (CFS), for example.

The LWK scheduler's primary task is to give as many CPU cycles as possible to a single task. Fairness and interactivity are not very important. This is in stark contrast to the quality of service that an FWK attempts to achieve.

2.2.3 Extreme simplification: Memory management

LWK memory managers are kept equally simple by providing a single policy, for a specific use, on a given architecture. Again, the goal for an LWK is to get out of the way. Instead of providing sophisticated memory allocation schemes and paging strategies to support a wide range of applications, large chunks (pages) of memory are handed to the application to use or misuse. A node running an LWK is allocated to a single user. Suboptimal use of that resource only impacts that user.

First-generation LWKs did not even have a single policy; merely a mechanism to provide an application with storage for text, data, and a large heap. Next-generation LWKs must support calls like **mmap()** and implement a physical memory management system.

2.2.4 Extreme simplification: Omit functionality

An LWK is also lightweight because it can omit a lot of functionality that FWKs and even small real-time and embedded kernels must have built in. This is possible because LWKs are paired with one or more FWKs in the system. That is a defining characteristic of an LWK. It needs a co-kernel or an FWK on another node to offload functions and receive other services. While the next-generation LWKs cooperate with an FWK on the same node, even traditional LWKs relied on the presence of an FWK on a nearby node: See Figure 2.

2.2.5 Extreme simplification: Space sharing

LWKs are designed for space sharing, while FWKs virtualize hardware by time sharing. LWKs can get away with that because they target massively parallel systems with lots of identical resources that are usually allocated in large numbers to a single user. Furthermore, supercomputers are designed to have specialized partitions for compute, I/O, and user interactions.

This simplifies the design and implementation of an LWK greatly. For example, a node in such a system represents a single security domain. While protection between processes is still desired, and the rest of the system must be protected from errant or malicious code, there are many security concerns that are much less important or can be ignored entirely. An FWK running in a large data center must support many different users on each server.

2.2.6 Extreme simplification: Code and binary size

While some characteristics of an LWK are easy to assess: "does it have a file system?", others are harder to quantify: "is it nimble?" Simplistically, many of the characteristics we have described are directly related, or expressed by, how much source code they require or binary code they execute.

Counting lines of source code is fraught with traps and pitfalls. It is not a good measure of complexity and not a useful metric for the definition of an LWK. Fewer lines of code *may* lead to higher performance and more nimbleness, but that alone is not sufficient for an LWK.

Comparing source code size to the code present in an FWK causes even more problems. An FWK solves a different problem than an LWK. It has to work well for an extremely diverse set of applications, platforms, and devices. An LWK is designed to run a small set of applications on a single platform very well.

An LWK is small in size – binary and source code – and

Table 1: Design goals of LWKs and FWKs

Design goal	LWK	FWK
Target	massively parallel systems	laptops, desktops, servers
Support	scalable applications	everything under the sun
Dev. environment for	parallel applications	business, games, commerce, etc.
Emphasis	efficiency	functionality
Resources	maximize use	fair sharing, QoS
Time to completion	minimal	when needed

should not be complex. It should be possible for a single person to understand and remember the entire LWK. This aids in debugging the kernel, adding new features, and pinpointing unexplained system behavior.

2.2.7 Coupling with an FWK

Linux supports tens of file systems, but most LWK have none. This is possible because LWKs work in cooperation with FWKs running on other nodes in the system. Most I/O functions and file system operations are function shipped to nodes running an FWK [5]. Next-generation LWKs ship to a Linux kernel that is local to the node: FusedOS [22], McKernel [24], and *mOS* [29] are examples. High-performance I/O still goes off-node, however.

2.2.8 Effects

There are several effects that result from these extreme simplifications.

Nimbleness: In addition to supporting highly-parallel applications, an LWK is also meant to adapt to new hardware and architectural features quickly. Extreme-scale systems often introduce new features that only gradually trickle to servers and desktop systems. Examples include high-performance networks and Remote Direct Memory Access (RDMA). Since the code base and complexity of an LWK are small, it can be nimble and adapt quickly to new requirements.

Portability: It turns out that porting LWKs is not actually that easy. The main reason for that is that they are often written for one, very specific, hardware architecture and take advantage of any specialized feature that makes sense. On the other hand, porting an LWK is not daunting, since it is small.

Behavior under load: FWKs are designed to gracefully degrade as load increases. LWKs have no degradation at all until load exceeds the capacity of the underlying hardware. This is not acceptable behavior in a desktop system or a sever environment. Therefore, that is something that would be difficult to achieve in an FWK.

Not having to degrade gracefully allows an LWK to exhibit much more consistent, deterministic performance. This is necessary for tightly coupled HPC applications to achieve scalability.

Layered services and composability: Most LWKs are designed with layered services and composability in mind. The idea is to have the ability to layer other services on top of the LWK. Supporting a full set of services should be enabled and could be implemented efficiently using an LWK.

Although this notion comes up frequently, few LWK implementation have succeeded with that. Microkernels seem to be doing a better job in this area.

2.3 Comparison to microkernels

A reasonable question to ask at this point is whether an LWK is a microkernel or maybe an exokernel.

We already mentioned in Section 2.2 one contrast between LWKs and microkernels. The former is meant to support a limited set of very specialized applications, while the latter is meant to provide the same functionality as an FWK, albeit using a different implementation approach. The difference of an LWK to a FWK or a microkernel is not in its implementation. From an LWK point of view, FWK and microkernels are no different: LWKs target specific machines and applications.

Composability is often mentioned in the context of LWKs and microkernels: The idea to tailor an OS or its services to a specific application or even a given execution and corresponding data set. An argument can be made that composability would allow the configuration of a microkernel that is suitable for extreme-scale computing. But, just like FWK, this would be bending a kernel into a use case that it was not really designed for.

A microkernel differs from an FWK mostly in its implementation. It targets the same users and applications as an FWK and has to live with the same conflicts to satisfy a diverse set of uses and environments. Although the kernel itself may be small, the services and their interactions, which are needed for a functional environment, can be very complex.

It is the goals and attributes that define an LWK; not the kernel size or its composability alone.

3. NEXT-GENERATION LWK

In the *mOS* architecture there are two different kernels running on the cores of a single node. One is a Linux kernel; the other is an LWK. This is different from traditional LWKs and we use the intersection to define an LWK. There are several projects that look at combining Linux functionality with an LWK. Kitten/Palacios, Fused OS, McKernel, and *mOS* are driven by anticipated usage models of future machines [20, 4]. The projects differ in how they provide Linux functionality while achieving LWK performance and scalability.

In Figure 2 we saw the components that make up a traditional LWK. The *mOS* LWK, as shown in Figure 3, is simpler. It can achieve this by offloading a lot of the functionality to Linux with which it shares a node. If the HAL was small before, it almost entirely disappears now because Linux can take care of most of it.

A lightweight co-kernel may be a purer lightweight architecture than an LWK that is forced to manage an entire node. The question now is: how little can we get away with

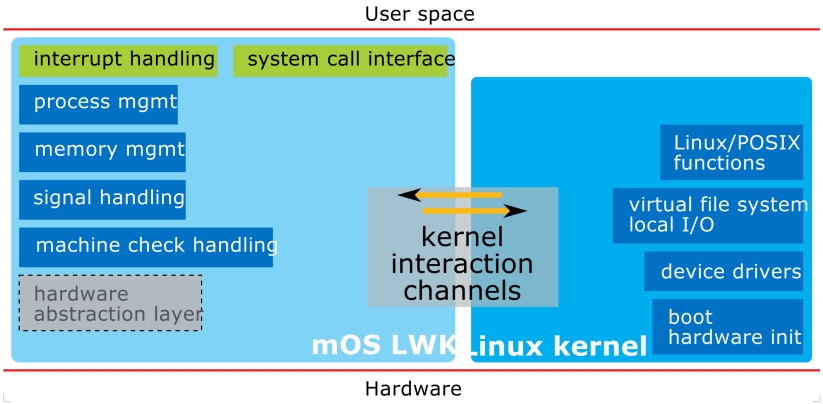


Figure 3: mOS LWK components and the Linux components which it uses

and still have an LWK?

Traditional LWKs support a subset of services to stay lightweight. In that regard, the *mOS* LWK is more like a microkernel in that it delegates services to another domain. In the case of McKernel and *mOS*, the LWK uses Linux as a server. This is still function shipping, but because the two kernels are now co-located, certain things become possible that were not before. For example, it is now feasible, at least in principle, to function-ship a file-backed **mmap()** call and share the work between the two kernels: Linux handles the file I/O portion of it, while the LWK deals with allocating the necessary memory.

Note that for Figure 3 we moved the block with "Linux / POSIX emulation" in Figure 2 to the right into the Linux kernel and renamed it. Emulation is no longer necessary: It *is* Linux executing on the other core(s).

As mentioned before, we hope that Linux will boot and initialize the node, as well as deal with its devices. Therefore, we use existing code in Linux for that and the corresponding boxes disappear from the LWK side of the picture.

The HAL is still on the LWK side, but we grayed it out. Many of those functions will no longer be needed, since there is so little code left in the LWK that deals directly with the hardware.

There is a new piece of code that was not there before: The kernel interaction channels are the means for the LWK to interact with Linux. There are different ways of achieving this. One is a proxy model, used by McKernel, where for each LWK process there also exists a process on the Linux side. That proxy makes the Linux system calls when instructed by the LWK, and transfers the results back to the LWK process. Another way to achieve this is letting the LWK call directly into the Linux kernel. These two approaches are contrasted in [10]

While the *mOS* LWK should be smaller and less complex than a traditional LWK, there are some added functions to be able to share a node with Linux, to interact with Linux, and to provide a high level of Linux compatibility.

It is fun to look at the concerns OS designers had more than forty years ago [3], contrast them to the design goals of LWKs in the nineties [28], and come to grips with what will be needed in the future.

Because the latest generation of LWKs share a node with Linux, performance isolation becomes an important goal in these co-kernel architectures [21, 15] and must be added to the design goals from Section 2.2.1.

For *mOS* there are additional design constraints that are now important but played no role in earlier LWKs. *mOS* must provide a high level of Linux compatibility. This must be achieved with as little intrusion as possible; i.e., no major or extensive changes to the Linux kernel. The system must remain maintainable and track future Linux developments.

mOS and McKernel run side-by-side with Linux on a node. Projects like Kitten with Palacios are pursuing a different approach. They make the LWK the hypervisor of a node and run Linux inside a virtual machine.

4. A DEFINITION

It is now time to create a clear, concise definition. So far we have learned that an LWK is defined by its design goals and attributes. Among them these seem most important:

- It targets a very specific set of machines and application types
- It relies on an FWK nearby
- There is no graceful degradation under load
- Its components and functionality are extremely simplified

Based on these observations, we could define an LWK like this:

> *"An LWK is a special-purpose OS kernel designed to support highly-scalable parallel applications. LWKs typically use simple resource management policies (e.g., static memory layouts, little or no time-sharing), provide direct user-level access to network hardware (OS bypass), and offload complex OS functionality to elsewhere (e.g., forwarding I/O calls to a dedicated server). A key design goal is to execute the target workload – highly-scalable parallel applications with non-trivial communication and synchronization requirements – with higher performance and more repeatable performance than is possible with a general-purpose OS approach."*

The main problem with this definition is that it is not very concise or lightweight. Maybe a constructive definition, as in [16], is more appropriate:

> *"A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."*

While this is true for microkernels and traditional LWKs, it is not sufficient for modern LWKs. Since they work in conjunction with an FWK, it is possible to offload *all* functionality to the FWK and achieve zero size. Which brings us to a quote by Antoine de Saint Exupéry:

> *"Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away."*

With that in mind, we declare:

> *"An LWK provides extremely simplified, bespoke services to highly-scalable, massively parallel applications. An LWK is deterministic in its behavior and delivers maximal performance and scalability, while relying on an FWK for additional functionality."*

5. RELATED WORK

Although each LWK project defines what they are doing, we are not aware of an all-encompassing LWK definition. In particular, none are sufficient to include the latest crop of LWKs.

Of course, there is no shortage in projects that aim to achieve LWK goals in practice. Some of them started with a Linux kernel and tried to tame it for HPC. Examples include ZeptoOS [1] and Cray's®CLE [13, 2]. Then there are traditional LWKs like SUNMOS [12, 17], Puma [27, 25], Catamount [14], and CNK [11] which struggled to achieve full Linux compatibility. More modern versions like Kitten together with the Palacios Virtual Machine Monitor [21], Fused OS [22], McKernel [24], and *mOS* [29] try to bring these worlds together.

This is not the first time this has been tried. Right-weight kernels [19] have also looked at extracting the essence of an LWK and fuse it into an FWK.

While early LWKs' sole purpose was to make supercomputing resources available to applications, newer variants are used to research and address problems specific to extreme scale. An example is [7] which makes use of the "smaller memory footprint, largely deterministic state, and simpler system structure" of an LWK to protect the OS from soft errors.

6. CONCLUSIONS AND FUTURE WORK

Groups working on LWKs usually have specific targets in mind and are less worried about a precise definition of what they are creating. Communicating about their research has been possible because participants in this corner of OS research "know an LWK when they see one."

Modern LWKs are harder to "see" because they are even smaller; relying on a co-kernel for most of the functionality needed. In the case of *mOS* this is even more true because the current, experimental, prototype embeds the LWK inside the Linux kernel.

The definition of an LWK needs to be refined further and will undoubtedly evolve over time. A survey article with a taxonomy would be of great assistance with this. In particular, differentiation to microkernels needs to be made more clear and work with other small and specialized OSes needs to be taken into consideration. This may prove useful not just from a research point of view but may have a positive influence on further development of LWKs.

Acknowledgments

This paper started out as a community effort to define what we mean by an LWK and help us explain it to people outside this subfield of expertise. This explains the long author list. Several other people helped shape this paper and its content. We thank: David van Dresser, Evan Powers, Kurt Alstrup, Lance Shuler, Larry Kaplan, Masamichi Takagi, Steve Hampson, Tom Musta, Torsten Hoefler, and Kamil Iskra.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

[1] ZeptoOS: The small Linux for big computers. http://www.mcs.anl.gov/research/projects/zeptoos/.

[2] Cray Linux Environment™ (CLE) 5.2 software release overview. `http://docs.cray.com/books/S-2425-52xx/S-2425-52xx.pdf`, Apr. 2014.

[3] Abernathy, D. H., Mancino, J. S., Pearson, C. R., and Swiger, D. C. Survey of design goals for operating systems. *SIGOPS Oper. Syst. Rev. 7*, 2 (Apr. 1973), 29–48.

[4] Akkan, H., Ionkov, L., and Lang, M. Transparently consistent asynchronous shared memory. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2013), ROSS '13, ACM.

[5] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable i/o forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on* (Aug 2009), pp. 1–10.

[6] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., and Wisniewski, R. W. Libra: A library operating system for a JVM in a virtualized execution environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 44–54.

[7] Ferreira, K. B., Pedretti, K., Brightwell, R., Bridges, P. G., Fiala, D., and Mueller, F. Evaluating operating system vulnerability to memory errors. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2012), ROSS '12, ACM.

[8] Gerofi, B., Shimada, A., Hori, A., and Ishikawa, Y. Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (May 2013), pp. 360–368.

[9] Gerofi, B., Shimada, A., Hori, A., Masamichi, T., and Ishikawa, Y. CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 73–84.

[10] Gerofi, B., Takagi, M., Ishikawa, Y., Riesen, R., Powers, E., and Wisniewski, R. W. Exploring the design space of combining linux with lightweight kernels for extreme scale computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (June 2015).

[11] Giampapa, M., Gooding, T., Inglett, T., and Wisniewski, R. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for* (2010).

[12] Greenberg, D. S., Maccabe, B., Riesen, R., Wheat, S., and Womble, D. Achieving high performance on the Intel Paragon. In *Proceedings of the Intel Supercomputer Users' Group. 1993 Annual North America Users' Conference* (Oct. 1993), pp. 203–208.

[13] Kaplan, L. Cray CNL. FastOS PI Meeting and Workshop, June 2007.

[14] Kelly, S., and Brightwell, R. Software architecture of the lightweight kernel, Catamount. In *Cray Users' Group Annual Technical Conference* (Albuquerque, New Mexico, June 2005).

[15] Kocoloski, B., and Lange, J. HPMMAP: Lightweight memory management for commodity operating systems. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 649–658.

[16] Liedtke, J. On micro-kernel construction. *SIGOPS Oper. Syst. Rev. 29*, 5 (Dec. 1995), 237–250.

[17] Maccabe, A. B., McCurley, K. S., Riesen, R., and Wheat, S. R. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference* (June 1994), pp. 245–251.

[18] Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H. Top500: The list. `http://www.top500.org`, Nov. 2014.

[19] Minnich, R. G., Sottile, M. J., Choi, S.-E., Hendriks, E., and McKie, J. Right-weight kernels: An off-the-shelf alternative to custom light-weight kernels. *SIGOPS Oper. Syst. Rev. 40*, 2 (Apr. 2006), 22–28.

[20] Otstott, D., Evans, N., Ionkov, L., Zhao, M., and Lang, M. Enabling composite applications through an asynchronous shared memory interface. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014* (2014), pp. 219–224.

[21] Ouyang, J., Kocoloski, B., Lange, J., and Pedretti, K. Achieving performance isolation with lighweight co-kernels. In *Proceeding of the 24th International ACM Symposium on High Performance Distributed Computing (HPDC)* (June 2015).

[22] Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D., and Wisniewski, R. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on* (Oct 2012), pp. 211–218.

[23] Riesen, R., Brightwell, R., Bridges, P. G., Hudson, T., Maccabe, A. B., Widener, P. M., and Ferreira, K. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience 21*, 6 (Apr. 2009), 793–817.

[24] Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A., and Ishikawa, Y. Interface for

heterogeneous kernels: A framework to enable hybrid OS designs targeting high performance computing on manycore architectures. In *High Performance Computing (HiPC), 2014 21th International Conference on* (Dec 2014), HiPC '14.

[25] Shuler, L., Jong, C., Riesen, R., van Dresser, D., Maccabe, A. B., Fisk, L. A., and Stallcup, T. M. The Puma Operating System for Massively Parallel Computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference* (1995), Intel Supercomputer User's Group.

[26] Soma, Y., Gerofi, B., and Ishikawa, Y. Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2014), ROSS '14, ACM.

[27] Wheat, S. R., Maccabe, A. B., Riesen, R., van Dresser, D. W., and Stallcup, T. M. PUMA: An operating system for massively parallel systems. *Scientific Programming 3* (1994), 275–288.

[28] Wikipedia. Lightweight kernel operating system. `http://en.wikipedia.org/wiki/Lightweight_Kernel_Operating_System`, Feb. 2015.

[29] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2014), ROSS '14, ACM, pp. 2:1–2:8.