

# N-way Light Weight Kernel Straw-man

Kevin T. Pedretti

November 20, 2006

## 1 Introduction

The past several years have seen a marked slowing in the rate of single processor performance improvement. The technical reasons for this have been eloquently stated many times elsewhere. To compensate, the industry has turned to utilizing multiple processors to drive mainstream computer performance improvements. Intuitively, a computer with two processors should be able to perform more work per unit time than a computer with one processor, assuming each individual processor has the same performance. This is not a concept we are unfamiliar with. What we are unfamiliar with is how to utilize single computers, aka compute nodes, with *lots and lots* of processors—in a couple of years, even a single socket is expected to contain 16 or more processors, each being multi-threaded. Our custom “light-weight kernel” (LWK) compute node operating system was not designed to handle this many processors in a single OS image. It may be possible to make our existing design work on some level, however, this author believes that a new LWK design and implementation cycle is called for. This paper presents a “straw-man” design for a light-weight kernel designed specifically for massively multi-core compute node. Many details are missing. The intention is to get people thinking and start to get some feedback. The overall goal is to end up with a LWK design that meets Sandia’s needs going forward and also has a chance of making an impact on a broader community.

The remainder of this paper is organized as follows.

## 2 Requirements

The overall N-way LWK requirements are:

1. Deterministic performance
2. Expose maximum performance of hardware to applications
3. Support for physically contiguous virtual memory
4. Debugger support
5. Performance counter support
6. Small manageable codebase of low complexity

The first two items go hand-and-hand. Having deterministic but slow performance is not acceptable. It is also not acceptable for performance to degrade with time, as is often the case with general purpose OSES such as Linux due to memory fragmentation. The baseline for comparison is Catamount running on Red Storm at scale. The third requirement is necessary to efficiently support network interfaces without address translation capability, such as Cray's SeaStar. It is closely related to the first two requirements, but must be explicitly stated. The fourth and fifth items are features supported by general purpose kernels and are expected by users. It is desirable that these be implemented to be compatible with existing tools and interfaces (e.g., GDB and Totalview debuggers, PAPI performance counter API). The last item is an implementation goal that is somewhat difficult to quantify. There is a tradeoff between functionality and complexity. It is expected that the N-way LWK will have a much smaller codebase than Linux but a somewhat larger codebase than Catamount's QK kernel.

In addition to the above general requirements, there is a set of features that are desired to meet the needs of a larger audience:

1. Code licensed with GPL (implies not export controlled)
2. Support for threads (processes that share an address space)
3. PCI support
4. Ethernet driver(s)
5. On-node RAM disk filesystem
6. Support for shared libraries
7. Security model allowing multiple users to share a compute node

First, having the code licensed under the GPL from the start is essential. Catamount is closed source and export controlled, severely limiting its audience and our potential to benefit from community contributions. Using the GPL license also allows for drawing code from Linux, which is also licensed under the GPL. The use of the GPL apparently causes no issues for vendors like Cray, since they are planning to use it as their compute node OS of choice going forward.

Second, supporting threads, or processes that share an address space, is one of the basic features needed to efficiently support non-MPI programming models such as OpenMP. If the amount of memory per socket decreases in the future, as some expect, it may become inefficient to run one MPI process per processor. This may make a hybrid programming model necessary. It is desirable that the N-way LWK interfaces for spawning threads be compatible with Linux's APIs, since this will enable compiler runtime libraries to be used without modification (e.g., a closed-source OpenMP library).

Third, a formalized PCI subsystem will make supporting new devices easier. Ad-hoc methods (i.e., hard coding) are currently used to interface with PCI devices (e.g., SeaStar). Note that PCI has become the de-facto standard for controlling devices. Even devices not connected to a PCI "bus" are made to look like PCI devices to the operating system. The main motivation for this,

besides standardization, is to enable general purpose OSes that have existing PCI support to support new interfaces (e.g., HyperTransport, PCI-Express) without modification.

Fourth, it is desirable to have some level of support for ethernet devices. This would simplify early-stage development. A significant source of trouble during early Red Storm development was creating a development platform to develop on before hardware was ready. The solution that was chosen was to rely on the BIOS Ethernet driver for network communication, which required some interesting acrobatics to utilize (e.g., “down-shifting” into real mode for every packet sent/received). It would have been much better if the LWK had had direct support for Ethernet devices. Ethernet support would also allow others to more easily try-out the N-way LWK on their existing systems (assuming an Ethernet driver existed for their hardware).

The fifth item probably raises some red flags—LWKs don’t do filesystems! However, it is often convenient to stage files into and out of a compute node. So long as the files reside fully in memory and the user can control how much memory is dedicated to files, performance can not be affected. The increase in kernel code complexity makes several tasks easier and/or more efficient. For example, Catamount currently fans out a timezone file needed by Glibc to every application process on a node. While this file is small, memory is used inefficiently because there is one copy per core, vs. one copy for the entire node if it was stored in a global RAM disk. The fan-out is also hard coded in the application startup code. It would be more flexible to allow users to specify which files are staged in/out. An on-disk RAM disk would also make executable code sharing more straight-forward. A staged-in ELF file on the RAM disk could be mapped into several address spaces without using any additional memory. If there were multiple memory controllers in a compute node, hence NUMA properties, the ELF file could be replicated onto a RAM disk in each memory controller’s memory, thus ensuring that instruction cache misses are always fetched from local memory.

The sixth item, shared library support, is a commonly requested feature by users. We protest that they aren’t scalable, however, with a sufficiently limited scope, they could be. Usually, an application will use a handful of shared libraries. These could be efficiently fanned-out to the compute nodes’ RAM disks before launching the application(s) requiring them. One appealing benefit of shared libraries in the context of multi-core is that they can potentially reduce code duplication. Consider an 80 processor compute node where each processor runs a different binary, but each binary utilizes the same library. Without shared library support, the common library will be duplicated in memory 80 times (assuming each binary uses all of the libraries functions). With shared library support, one copy of the library could be stored in memory and each binary could map it into its address space. Note that there is no benefit if all 80 cores run the same binary since there would be only one copy of it stored in memory.

Finally, the LWK design should not rule out the possibility of sharing a compute node between multiple users. While this usually doesn’t make sense for capability systems (an exception might be interactive debug nodes), smaller systems often require this capability.

- 3 High-level Overview**
- 4 Memory Management**
- 5 Process Management**
- 6 Interrupts and System Calls**
- 7 On-node Filesystem**
- 8 External Interface**
- 9 Development Plan**
- 10 Conclusion**

Get foundation right (process management, mem management, interrupts).  
Then focus on other subsystems (multiple efforts in parallel).