Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing

Balazs Gerofi†, Takagi Masamichi†, Yutaka Ishikawa†, Rolf Riesen‡, Evan Powers‡, Robert W. Wisniewski‡

†RIKEN Advanced Institute for Computational Science, Japan
‡Intel Corporation, US
Outline

• Motivation
• Background
  • HPC OS Architecture and Lightweight kernels (LWK)
• Linux + LWK
  • Issues and Requirements
  • Design
    • Proxy model
    • Direct model
• Comparison
• Evaluation
• Conclusion
Motivation

• Complexity of high-end HPC systems keeps growing
  • Extreme degree of parallelism
  • Heterogeneous core architectures
  • Deep memory hierarchy
  • Power constrains

Need for scalable, reliable performance and capability to rapidly adapt to new HW

• Applications have also become complex
  • In-situ analysis, workflows
  • Sophisticated monitoring and tools support, etc..
  • Isolated, consistent simulation performance

Dependence on POSIX and the rich Linux APIs

• Seemingly contradictory requirements..
• Is the current system software stack ready for this?
Background – HPC Node OS Architecture

- Traditionally: driven by the need for scalable, consistent performance for bulk-synchronous HPC

```
Linux
- Full Linux API
  - TCP stack
  - VFS
  - Dev. Drivers
  - FileSys Drivers
  - General Scheduler
```

- Start from Linux and remove features impeding HPC performance
- Eliminate OS noise (daemons, timer IRQ, etc.), simplify memory management, simplify scheduler

“Stripped down Linux” approach
(Cray’s Extr. Scale Linux, Fujitsu’s Linux, ZeptoOS, etc..)

```
HPC OS
- Simple Mem Mngt.
- Proc Mngt.
- Simple Scheduler
```

No full Linux API!
Background – HPC Node OS Architecture

• Traditionally: driven by the need for scalable, consistent performance for bulk-synchronous HPC

• Start from a thin Light Weight Kernel (LWK) written from scratch and add features to provide a more Linux like I/F, but keep scalability

• Support dynamic libraries, allow thread over-subscription, support for /proc filesys, etc..

No full Linux API!

“Enhanced LWK” approach (Catamount, CNK, etc..)
Outline

✓ Motivation
✓ Background
  ✓ HPC OS Architecture and Lightweight kernels (LWK)

• Linux + LWK
  • Issues and Requirements
  • Design
    • Proxy model
    • Direct model

• Comparison
• Evaluation
• Conclusion
Hybrid Linux + LWK Approach

• With the abundance of CPU cores a new hybrid approach: run Linux and LWK side-by-side!
• Partition resources (CPU core, memory) explicitly
• Run HPC apps on LWK
• Selectively serve OS features with the help of Linux by offloading requests
Hybrid Linux + LWK Approach

- With the abundance of CPU cores a new hybrid Linux + LWK side-by-side!
- Partition resources (CPU core, memory) explicitly
- Run HPC apps on LWK
- Selectively serve OS features with the help of Linux by offloading

Where is the border between the two kernels?

What features are implemented in the LWK?

How to integrate the two types of kernels?

Where should tools run and how do they interact with apps?
Linux + LWK: Requirements

• **Scalability and performance:**
  • System has to deliver LWK scalability, reliability and consistent performance

• **Linux compatibility:**
  • Support for POSIX and Linux APIs is an absolute must

• **Adaptability (a.k.a., nimbleness):**
  • System should seamlessly adapt to new HW features and SW needs

• **Maintainability:**
  • System should be highly maintainable, esp. tracking Linux changes
Linux + LWK: the Proxy Model

- LWK is completely independent from Linux
- Proxy process: serves as execution context for offloaded system calls
- Ensures that Linux kernel maintains necessary state information about the application (i.e., file desc. table)
IHK/McKernel: an implementation of the Proxy Model

- **Interface for Heterogeneous Kernels (IHK):**
  - Low level software infrastructure
  - Allows partitioning SMP chips (using the Linux hotplug service)
  - Enables management of resources and light-weight kernels
    - Create OS instances, assign resources
    - Load kernel images, boot OS, etc..
  - Provides low-level Inter-Kernel Communication (IKC) layer

- **McKernel:**
  - A lightweight kernel designed for HPC
  - Booted from IHK and requires Linux’ presence
  - Only performance sensitive syscalls (MM, PM, perf counters) impl.
    - Rest offloaded to Linux
  - Simple co-operative round-robin scheduler
IHK/McKernel: unified address space

- System call offloading: what to do with pointers?
  - Linux kernel may access them (i.e., copy_from_user(), etc.)
- User-space virtual to physical mappings are set up to be the same in Linux so that the proxy process can access syscall arguments
- A pseudo file mapping in mcexec (proxy process) covers the entire user-space of McKernel. When page fault occurs we trap the handler and set up mappings so that they point to the same physical pages
Linux + LWK: mOS’ Direct Model

- Tight integration between LWK and Linux
- LWK passes kernel data structures to Linux when offloading system calls (Evanescence)
  - i.e., migrates the task_struct
- LWK is “compiled-in”
- Anticipated: various kernel features will be available directly (e.g., no need to deal with pointers in syscall offloading, etc.)
- CPU cores and memory are isolated but Linux is aware of them, Linux IRQs are not directed to LWK cores
- Which OS services are implemented in LWK and to what extent the LWK implementation is restricted are somewhat unclear
Outline

✓ Motivation
✓ Background
  ✓ HPC OS Architecture and Lightweight kernels (LWK)
✓ Linux + LWK
  ✓ Issues and Requirements
  ✓ Design
    ✓ Proxy model
    ✓ Direct model

• Comparison
• Evaluation
• Conclusion
Comparison: Linux compatibility

- POSIX and Linux API compatibility on LWK:
  - Correct syscall API and execution (involving offload)
  - Valid signaling behavior
  - Availability of `/proc` and `/sys` statistical information
  - Availability of the `ptrace` interface (many tools rely on this)
  - Virtual dynamics objects (VDSO) page
  - Performance counters (PAPI, etc.)

- Proxy model (McKernel):
  - Significant implementation effort because kernel code is isolated
  - `/proc` and `/sys`:
    - Redirections from the syscall offload kernel module
    - Data needs to be obtained from McKernel
  - How far should McKernel present itself as “stand alone”? `/proc/cpuid`, etc.

- mOS:
  - Lot of things are expected to “fall out” for free
  - `/proc` and `/sys` can be directly accessed due to shared kernel structures, VDSO just works
  - Will signaling/ptrace work?
    - Cross-kernel IPIs?
Comparison: Flexibility

- LWK Flexibility (a.k.a., nimbleness)
  - Ability to easily adapt to new requirements
  - Rapid prototyping for unconventional hardware
    - Heterogeneous cores, deep memory hierarchies, multiple cache coherence domains, etc..
  - Easy experimentation for exotic runtimes
  - Small LWK codebase is favored

- McKernel:
  - Standalone kernel code with small codebase
    - IHK+McKernel: ~50k lines
  - Full control over implementation
  - Allows proprietary kernel images as well

- mOS:
  - Tighter integration with Linux (data structures) could limit flexibility
    - Translation between mOS and Linux kernel structures before and after syscall offloading could alleviate the issue
Comparison: Linux modifications

• Required Linux modifications
  • It’s hard to keep track upstream Linux changes
  • Keeping the number of changes minimal is first priority

• IHK/McKernel:
  • No changes to Linux kernel but unexported symbols are accessed
    • irq_desc_tree, wakeup_secondary_cpu_via_init(), etc..

• mOS:
  • Evanescence requires some changes (~150 lines) to Linux kernel
    • System call wrapper macros for offloading
  • mOS’ objective is to implement most LWK functions isolated from the Linux codebase
Comparison: LWK reboot

• LWK reboot capability
  • Reboot can provide a clean, well defined state
  • Allows easy deployment of different/customized LWK kernel images
  • Rebooting the LWK between jobs can lead to more predictable application behavior
  • “Reboot every time” design may simplify the LWK, no need for concepts such as switching between apps/users

• Proxy model (McKernel):
  • Warm rebooting LWK cores is supported
  • Can solve problems where otherwise rebooting the node would be required (inconsistency in LWK, etc.)

• mOS:
  • LWK cores do not go through their own trampoline code → no rebooting in its strict sense
  • CPU hotplug/re-plug may be used
Outline

✓ Motivation
✓ Background
  ✓ HPC OS Architecture and Lightweight kernels (LWK)
✓ Linux + LWK
  ✓ Issues and Requirements
  ✓ Design
    ✓ Proxy model
    ✓ Direct model
✓ Comparison
  • Evaluation
  • Conclusion
Evaluation

- **HW:** Ivy Bridge (E5-2670) – same LWK CPU cores in each measurements
- **Test:** simple “empty” syscall offloading cost comparison

Surprisingly similar performance!
- mOS offload performance will probably improve in the future
- IHK/McKernel plans for evaluating MONITOR/MWAIT
Conclusion

• Need for enhanced system software to meet HW and SW requirements of exascale and beyond
• Hybrid Linux+LWK approach proposed to provide LWK performance but to retain Linux APIs at the same time
• Explored design alternatives and their trade-offs

• Direct model (mOS @ Intel)
  • Less development effort for Linux compatibility and tools support

• Proxy model (IHK/McKernel @ RIKEN)
  • Retains full control over the code base of the LWK and thus has a higher degree of flexibility
Thank you for your attention! Questions?