A Design of Hybrid Operating System for a Parallel Computer with Multi-Core and Many-Core Processors

Mikiko Sato\textsuperscript{1,5}, Go Fukazawa\textsuperscript{1}, Kiyohiko Nagamine\textsuperscript{1}, Ryuichi Sakamoto\textsuperscript{1}
Mitaro Namiki\textsuperscript{1,5}, Kazumi Yoshinaga\textsuperscript{2,5}, Yuichi Tsujita\textsuperscript{2,5}, Atsushi Hori\textsuperscript{3,5}
Yutaka Ishikawa\textsuperscript{3,4}

\textsuperscript{1}Tokyo University of Agriculture and Technology \hspace{1cm} \textsuperscript{2}Kinki University
\textsuperscript{3}RIKEN Advanced Institute for Computational Science \hspace{1cm} \textsuperscript{4}University of Tokyo
\textsuperscript{5}Japan Science and Technology Agency, CREST
Outline

• Background
• Motivation
• Design of Multi-core & Many-core system
  ➢ Hardware architecture
  ➢ Software architecture
• Prototype system and evaluation
• Conclusion
Background of this study

• Supercomputers are predicted to achieve exaflop performance by 2018.

What kind of the system architecture can achieve it?

• Node architecture of the supercomputer
  ➢ Multi-core processor nodes
    • Sequoia, Mira, Fermi, JuQUEEN (PowerBQC 16C)
    • K computer (SPARC64 VIIIfx)
    • SuperMUC (Xeon E5-2680 8C)
    • Jaguar (Opteron 6274 16C)
  ➢ Multi-core and GPGPU nodes
    • Tianhe-1A (Xeon+NVIDIA 2050)
    • Nebulae (Xeon+NVIDIA 2050)
    • TSUBAME 2.0 (Xeon, NVIDIA 2050)
Our target

- Multi-core and Many-core processor combination
  - Intel Many Integrated Core (MIC)
    - Xeon Phi (Knights Corner)
    - multiple simple X86 cores ( > 50 cores )
  → Plan to apply MIC to the computing node to achieve highly parallel processing!
Motivation and Approach

• The reason of choosing MIC

➢ Compatibility with x86 programming models
  → the *System software* has to support APIs on *MIC*
    – UNIX/POSIX-API (memory management, I/O access etc.)
    – Parallel programming API (MPI/Thread etc.)

➢ MIC is possible to run OS

MIC has few memories per core (8GB RAM are used by >50 cores)

→ The Operating System on MIC should be **light weight!**
  – Host OS performs some functions instead of light weight OS.

**What kind of Host OS supports are effective for LWOS?**
(process, memory, and I/O management)
Hybrid computer system overview

- Linux on Multi-core CPU works as Host OS
  - I/O devices and Many-core resources management
- Light Weight OS on Many-core CPU (MIC)
  - Thread management with low noise

similar to recent LWK (CNK, Kitten, Catamount)

computing node

- Multi-core CPU
  - core1, core2, core3, core4
- Host OS (Linux)
- Host System Application Programs
- Inter-OS Collaboration Mechanism
- Parallel Programs
- Shared memory
- PCIe ( or QPI ?)
- Low noise for high performance.

Front-end node (distribute tasks)
Process model

- Host-process and LW-processes are formed into a group (Task).
  - Host-process controls LW-process (create, delete, etc.)
  - POSIX Threads execute using the LW-process resources.
  - MPI is used for inter-process communication
    - The main node distributes the Task group to each node using MPI
LW-process Management

• Host OS roles
  (1) The Host OS manages the cores of the Many-core CPU and assigns free cores to the LW-processes
  (2) The Host OS manages the physical memory of the many-core processor and continuously assigns memory areas to LW-processes.
  (3) The Host OS manages I/O access requests from Threads on LWOS.

• LWOS role
  ➢ Thread management
    – LWOS delegates the resource management function to the Host OS.
Host OS structure (user-level)

- **LW-process management**
  - control the LW-process
    - (Create, Destroy, Suspend, Resume)
    - load the ELF-binary program
    - complete the page table
    - keep the LW-process context
      - Physical core on Many-core
      - Physical memory of Many-core
      - The start address of Page Table
      - Inter-OS comm. buffer address

- **Resource management**
  - assign physical resources
  - keep the free resources
Memory mapping for LW-process

(1) Assign the required continuous memory:
- LW-process segment
- Page table segment
- Inter-OS comm. buffer

(2) Map the memory region to Host-process address space

(3) Parse the ELF-binary program, create page table for address trans.

(4) Load the program (fully assign stack & heap)
Host OS structure (kernel-level)

I/O access support

- Receive I/O access requests (open, close, read, write, etc..)
  - Buffer address
  - File descriptor
  - Access size

- I/O access at Kernel module
  - change into virtual address on Host OS
  - Access through Linux file system

- Keep the file context
  - The information of opened file
  - Check it when accessing the file.
LWOS structure (User Level)

• POSIX Thread Library
  ➢ Each core on Many-core CPU is virtualized to a thread.
  ➢ Thread management
    – pthread_create, join, etc..
    – Non-preemptive execution
    – Keep the thread contexts

• POSIX API Delegation Library
  ➢ Set the arguments of POSIX API to Inter-OS communication buffer
  ➢ Notice the delegation to Host OS
    – via Kernel level IPI handler
LWOS structure (Kernel Level)

- LW-process Management
  - Receive LW-process control requests from Host OS (Create, Destroy, Suspend, Resume)
    - start the LW-process execution
    - Up-call to destroy/suspend/resume
  - Keep the LW-process context
    - The number of physical core and IDs
    - the start address of the page table (set it to the page directory base register)
    - the Inter-OS comm. buffer address
    - the entry point address of the program
Inter-OS communication

• Set the command items and arguments to the inter-OS communication buffer at User-level
  – I/O access datas are directly written to LW-process address space
• Send “IPI” interruption at Kernel-level
  → reducing the buffer copy overhead

<table>
<thead>
<tr>
<th>Table 1: LW-process management notification items.</th>
<th>Table 2: Memory management notification items.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>Items</td>
</tr>
<tr>
<td>LW-process Create</td>
<td>Page Fault Notice</td>
</tr>
<tr>
<td>LW-process Suspend</td>
<td>Page assign Notice</td>
</tr>
<tr>
<td>LW-process Resume</td>
<td></td>
</tr>
<tr>
<td>LW-process Stop</td>
<td></td>
</tr>
<tr>
<td>LW-process State Notify</td>
<td></td>
</tr>
<tr>
<td>Send to</td>
<td>Send to</td>
</tr>
<tr>
<td>LWOS</td>
<td>Host OS</td>
</tr>
<tr>
<td>LWOS</td>
<td>Host OS</td>
</tr>
<tr>
<td>LWOS</td>
<td>Host OS</td>
</tr>
<tr>
<td>LWOS</td>
<td>Host OS</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: I/O access notification items.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
</tr>
<tr>
<td>Send to</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Device Open</td>
</tr>
<tr>
<td>Host OS</td>
</tr>
<tr>
<td>Asynchronous</td>
</tr>
<tr>
<td>Device Close</td>
</tr>
<tr>
<td>Host OS</td>
</tr>
<tr>
<td>Asynchronous</td>
</tr>
<tr>
<td>Device Read</td>
</tr>
<tr>
<td>Host OS</td>
</tr>
<tr>
<td>Asynchronous</td>
</tr>
<tr>
<td>Device Write</td>
</tr>
<tr>
<td>Host OS</td>
</tr>
<tr>
<td>Asynchronous</td>
</tr>
</tbody>
</table>
LW-process management API

- **Host OS APIs**
  - Host process uses these APIs for a *LW-process control*

- **LWOS APIs**
  - *Memory and thread management* API are performed in LWOS without notifying to HOST OS.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwp_create, lwp_suspend, lwp_resume, lwp_destroy</td>
<td>LW-process control (to LWOS)</td>
</tr>
<tr>
<td>lwp_wait</td>
<td>Wait the LW-process exit (from LWOS)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>LW-process exit (to Host)</td>
</tr>
<tr>
<td>brk, sbrk</td>
<td>Management of heap memory</td>
</tr>
<tr>
<td>open, close, read, write, ioctl</td>
<td>Linux File I/O (to Host)</td>
</tr>
<tr>
<td>pthread_*</td>
<td>POSIX thread I/F</td>
</tr>
</tbody>
</table>
Prototype system for evaluation

- Two Xeon CPUs performed Multi-core and Many-core.
  - Confirm processing mechanism of LW-process creation
  - LWOS for Many-core CPU is under development
    - LWOS is executed on one core.
    - LW-process and LWOS library are implemented to kernel level.
Evaluation of LW-process creation

- measure the execution time of a program that creates the LW-process and then immediately terminates it.
  - compared with Linux fork-exec overhead (only advisory)

<table>
<thead>
<tr>
<th>environment</th>
<th>time</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed hybrid OS</td>
<td>110us</td>
<td>147%</td>
</tr>
<tr>
<td>Linux</td>
<td>75us</td>
<td>100%</td>
</tr>
</tbody>
</table>

Host-process Code
```
void main() {
    lwpid = lwp_create();
    lwp_wait(lwpid);
}
```

LW-process Code
```
void main() {
    exit();
}
```

binary size = 1KB
Analysis of the processing time

• The parse and load of the binary file use most time.
• The rate of Inter-OS communication overhead was low.
  ➢ we used internal bus on Xeon system in this evaluation.
    – communication speed is faster than real MIC environment.
  ➢ we will evaluate the processing rate on MIC environment.

- LW-process create on Host OS
  • parse and load the binary (most time)
  • create page table

- Inter-OS communication overhead

- other (3%)
Conclusion

• We have proposed a system in which the functions for managing the resources of many-core processors are delegated to the Host OS running on multi-core processor in a parallel computing system that uses both multi-core processors and many-core processors.
• That approach allows the LWOS that runs on a many-core processor to be dedicated to the execution of parallel computation programs.
• We have described the structures of the Host OS and LWOS and explained about inter-OS communication.
Future works

• LWOS implementation on MIC processor
  ➢ Thread management on many-core processor
  ➢ Evaluate LW-process management overhead and inter-OS communication overhead

• We will investigate many-core assignment algorithms on Host OS to improve the parallel computing performance of many-core processors.
Thank you
LW-process control APIs

lwp_create

- Binary file path, the number of core, Stack size, heap size
- return: process ID

lwp_suspen / lwp_resume

- process ID

lwp_destroy

- process ID

lwp_wait

- process ID

```c
void api_sample(void)
{
    /*
     * 4core, 1MB stack, 100MB heap
     */
    lwpid = lwp_create(~/calc_program.bin", 4, 0x100000, 0x6400000);
    
    /* suspend and resume*/
    lwp_suspend ( lwpid );
    lwp_resume ( lwpid );

    /* wait for exit*/
    lwp_wait ( lwpid );
}
```