Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support

Kyle C. Hale and Peter Dinda
Hybrid Runtimes

the runtime IS the kernel

runtime not limited to abstractions exposed by syscall interface

opportunity for leveraging privileged HW features
CURRENT OS/R MODEL

user-mode

- PARALLEL APP
- PARALLEL RUNTIME

kernel-mode

- GENERAL PURPOSE OS
- HARDWARE
THE HYBRID RUNTIME

user-mode

kernel-mode

kernel-mashup of OS and runtime
NESL: A Parallel Programming Language

NDPC

Racket

Legion

Los Alamos National Laboratory

NVIDIA
Speedup over Linux

Legion Processor Count (Cores)

Xeon Phi  x64

1.4
1.3
1.2
1.1
1

Speedup over Linux

Legion Processor Count (Cores)

Xeon Phi

x64

1.19x speedup over linux
The graph shows the speedup over Linux for Xeon Phi and x64 processors with varying Legion Processor Count (Cores). The speedup for Xeon Phi is indicated by the blue line, and for x64 by the orange line.

- Xeon Phi achieves a 1.37x speedup over Linux at a processor count of 64 cores.
- x64 achieves a 1.19x speedup over Linux at a processor count of 128 cores.

The graph illustrates how the speedup increases with the processor count, peaking before declining at higher processor counts.
TWO ENABLING TOOLS
nautilus

kernel framework
bridge HRT with legacy OS
OUTLINE

Background/Overview

Nautilus

Deployment Models

Hybrid Virtual Machine

Multiverse & Future Work
user-mode

kernel-mode

aero

kernel

HARDWARE

Nautilus primitives & utilities (HRT can use or not use any of them)
Nautilus

under the hood
Parallel Runtime System

Nautilus
kernel primitives should be SIMPLE and FAST

runtime developer can easily reason about them!
start with familiar interfaces

threads

condition variables

mutexes/locks

memory management

fork/join parallelism
map runtime’s logical view of machine onto physical HW

Parallel Runtime System

logical CPUs
threads

unified, shared address space

threads can operate preemptively or cooperatively*

*for runtimes that require more determinism
thread creation is **FAST**

**x86_64 Opteron:** 64 cores, 4 sockets, 8 numa zones, 128GB RAM
thread creation is **FAST**
thread creation is **FAST**

Cycles

- Nautilus
- Linux (pthreads)

~3ms

~90µs

x86_64 Opteron: 64 cores, 4 sockets, 8 numa zones, 128GB RAM
software events
user-mode software events are SLOW

lower is better

trigger latency (cycles)

cond. var.  futex  hardware (IPI)

x86_64 Opteron: 64 cores, 4 sockets, 8 numa zones, 128GB RAM
nautilus events triggers are FASTER

![Graph showing trigger latency comparison for different methods: cond. var., futex, naut. cond. var., naut. cond. var. w/IP, hardware (IPI). The y-axis represents trigger latency in cycles, ranging from 0 to 35,000 cycles. The x-axis lists the methods as cond. var., futex, naut. cond. var., naut. cond. var. w/IP, and hardware (IPI). The graph indicates that hardware (IPI) has the lowest trigger latency, followed by naut. cond. var. w/IP, naut. cond. var., futex, and cond. var. respectively.]

x86_64 Opteron: 64 cores, 4 sockets, 8 numa zones, 128GB RAM
TRIGGER LATENCY (CYCLES)

~100 cycles

IPI

Nemo
memory
static identity map

bootup

page size

physical memory

2MB

1GB
eliminates expensive page faults

reduces TLB misses + shootdowns

increases performance under virtualization
NUMA

runtime has FULL control over thread placement and memory layout
philix
philix host utility

host

Intel MPSS stack

virtual console

Xeon Phi

custom OS
Nautilus is fairly small

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nautilus</td>
<td>25,000 C</td>
</tr>
<tr>
<td>Legion</td>
<td>43,000 C++</td>
</tr>
<tr>
<td>Additions for Legion</td>
<td>800 C/C++</td>
</tr>
<tr>
<td>Additions for Xeon Phi</td>
<td>1350 C</td>
</tr>
</tbody>
</table>
What do we lose?
familiar environment
driver ecosystem
protection/isolation
OUTLINE

Background/Overview

Nautilus

Deployment Models

Hybrid Virtual Machine

Multiverse & Future Work
DEDICATED

APPLICATION

HRT

full HW access

HARDWARE
PARTITIONED

- LINUX STACK
- HARDWARE
- CORE 0 ... CORE K-1
- CORE K ...
- CORE N-1

- APPLICATION
- HRT
- full HW access
Hybrid Virtual Machine

- **HARDWARE**
- **HRT**
- **PARALLEL APP**
- **LINUX STACK**
- **VMM**
- **HVM**
- **regular OS vcores**
- **HRT vcores**
OUTLINE

Background/Overview

Nautilus

Deployment Models

Hybrid Virtual Machine

Multiverse & Future Work
regular OS (ROS)

PARALLEL APP

PARALLEL RUNTIME

GENERAL PURPOSE OS

NODE HARDWARE

GENERAL VIRTUALIZATION MODEL

HVM

legacy functionality from ROS via HVM

performance path

user-mode

kernel-mode

user-mode

kernel-mode

PARALLEL APP

HRT

NODE HARDWARE

SPECIALIZED VIRTUALIZATION MODEL
ROS/HRT setup

Virtual Machine
ROS/HRT setup

VM memory

Virtual Machine

ROS visible memory

ROS vcore

HRT vcore

ROS vcore

HRT vcore
merged address space

ROS vaddr space

higher half
ROS kernel

app + runtime
code & data

merged

Aerokernel

HRT vaddr space
ROS/HRT communication

Address space: Merge boot time request

HYPervisor + HVM

shared data page

HRT
ROS/HRT communication

- Synchronous operation request
- Shared data page
- Comm. area
- Hypervisor + HVM
## ROS/HRT communication

<table>
<thead>
<tr>
<th>Description</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr space merge</td>
<td>~33K</td>
<td>15µs</td>
</tr>
<tr>
<td>Asynch function invocation</td>
<td>~25K</td>
<td>11µs</td>
</tr>
<tr>
<td>Synch function invocation (remote socket)</td>
<td>~1060</td>
<td>482ns</td>
</tr>
<tr>
<td>Synch function invocation (same socket)</td>
<td>~790</td>
<td>359ns</td>
</tr>
</tbody>
</table>
HRT boot

HVM sets up
registers, in
systemcallблиbolksly

*no need for INIT-SIPI-SIPI sequence
LINUX FORK + EXEC \(\sim 714\mu s\)

HVM + HRT CORE BOOT \(\sim 61\mu s\)
how do we take a legacy runtime to the HRT + X model?

PORT
porting a runtime/app to an a new OS environment is...

DIFFICULT

TIME-CONSUMING

ERROR-PRONE
development cycle:

do { 

ADD FUNCTION

REBUILD

BOOT HRT

} while (HRT falls over)
much of the functionality is NOT ON THE CRITICAL PATH
we want to make this easier
give us your legacy

runtime
we automatically transform it to run as an HRT in kernel mode bridged with a legacy OS (AUTOMATIC HYBRIDIZATION)
rebuild with our toolchain
LEGACY APP/RUNTIME
MULTIVERSE RUNTIME LAYER
AEROKERNEL BINARY
KERNEL-MODE
LINUX
HYPERVISOR + HVM

runs like normal application

BOOTED AEROKERNEL

LEGACY APP/RUNTIME
# ls
bench-write.out  go  mracket-GOLD
binary-tree-2.rkt  intsum-native  multiverse-racket
bytes  isn  multiverse.log
collects  ism  nbody.rkt
doall.sh  lgn-hpcg  racket
doruns.sh  lgo  results
fannkuch-redux.rkt  lost+found  spectral-norm.rkt
fasta-3.rkt  lpm  test.out
fasta.rkt  lpn  test.t
g  mandelbrot-2.rkt
4 parallel runtimes (Legion, Racket, NDPC, NESL)

Hybrid Virtual Machine
bridge HRT with legacy OS

Multiverse
automatic transformation: legacy app+runtime → HRT

nautilus
thank you
my webpage: halek.co
lab: presciencelab.org
download nautilus: nautilus.halek.co
v3vee project: v3vee.org
backups
thread fork
interrupt driven execution
memory and paging
static identity map
eliminates expensive page faults

reduces TLB misses + shootdowns

increases performance under virtualization
Unified TLB Misses

$\log$ scale

10,000,000

1,000

1
thread creation is **PREDICTABLE**

Log scale

Lower is better

Cycles

Linux (pthreads) vs Nautilus

*x86_64 Opteron: 64 cores, 4 sockets, 8 numa zones, 128GB RAM*
binding to physical processor is GUARANTEED
RUNTIME control over physical CPU
interrupts
&
events
asynchronous notifications
trigger latency

EVENT COMPLETES

trigger

DEPENDENT TASK EXECUTES
user-mode software events are **SLOW**

lower is better

---

**x86_64 Opteron**: 64 cores, 4 sockets, 8 numa zones, 128GB RAM
**nautilus events triggers are FASTER**

![Box plot showing trigger latency with and without hardware (IPI)]

- **cond. var.**
- **futex**
- **naut. cond. var.**
- **naut. cond. var. w/IPI**
- **hardware (IPI)**

---

**x86_64 Opteron: 64 cores, 4 sockets, 8 numa zones, 128GB RAM**
handle_event()  handle_event()
asynchronous, IPI-based execution
NOT POSSIBLE IN LINUX USERSPACE
<table>
<thead>
<tr>
<th>TRIGGER LATENCY (CYCLES)</th>
<th>IPI</th>
<th>Nemo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1620</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1440</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>900</td>
<td></td>
<td></td>
</tr>
<tr>
<td>720</td>
<td></td>
<td></td>
</tr>
<tr>
<td>540</td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*lower is better*
TRIGGER LATENCY (CYCLES)

0 180 360 540 720 900 1080 1260 1440 1620 1800

IPI

Nemo
TRIGGER LATENCY (CYCLES)

- IPI
- Nemo

~100 cycles
this gives us an incremental path for creating HRTs
start out with a working HRT system
then pull functionality into the HRT for hot spots
RACKET

most widely used Scheme implementation

downloaded ~300 times/day

complex runtime with JIT
ROS/HRT communication

- Aerokernel Binary
- Legacy App/Runtime
- Multiverse Runtime Layer
- Linux System Call
- Booted Aerokernel
- Hypervisor + HVM

System Call: Legacy Runtime
Multiverse Runtime Layer
Broadcast wakeups on x64

Cycles to Wakeup

 pthread condvar  futex wakeup  Aerokernel condvar  Aerokernel condvar + IPI  broadcast IPI

- pthread condvar
  - min = 17538
  - max = 2.17277e+06
  - μ = 995795
  - σ = 544512

- futex wakeup
  - min = 16402
  - max = 1.89553e+06
  - μ = 370630
  - σ = 199680

- Aerokernel condvar
  - min = 3258
  - max = 612959
  - μ = 265820
  - σ = 159421

- Aerokernel condvar + IPI
  - min = 7842
  - max = 464015
  - μ = 132417
  - σ = 98637.4

- broadcast IPI
  - min = 1252
  - max = 57467
  - μ = 12827.3
  - σ = 2931.32

Nautilus events
Wakeup deviation on x64

Aerokernel condvar
Aerokernel condvar + IPI
pthread condvar
futex broadcast
broadcast IPI
Wakeup deviation on x64

CDF

HW lower bound

user-mode events

Nautilus events

pthread condvar
futex broadcast
Aerokernel condvar
Aerokernel condvar + IPI
broadcast IPI

2x
Unicast IPIs on phi

95th percentile = 746 cycles

CDF

Cycles measured from BSP (core 224)
CDF

95th percentile = 2105 cycles

Cycles measured from BSP (core 224)
Multicast IPIs on phi

95\textsuperscript{th} percentile $\sigma = 31$ cycles
Wake up deviation on x64

CDF

pthread condvar
futex broadcast
broadcast IPI

$s$
Wakeup deviation on x64

CDF

HW lower bound

user-mode events

~70x

pthread condvar
futex broadcast
broadcast IPI

\[
\sigma
\]

1000 10000 100000 1\times10^6
thread context switch on x64

![Graph showing thread context switch cycles for Linux (pthreads) and Aerokernel threads.]

- **Linux (pthreads)**
  - Min: 1352
  - Max: 2438
  - Mean (μ): 1412.67
  - Standard Deviation (σ): 114.377

- **Aerokernel threads**
  - Min: 1176
  - Max: 1391
  - Mean (μ): 1269.52
  - Standard Deviation (σ): 48.3902
thread create + launch on phi

Cycles

Linux (pthreads)  Linux (kernel threads)  Nautilus kernel threads
thread create + launch (many threads) on x64
thread create + launch
(many threads) on phi
Unicast IPIs on x64

CDF

95th percentile = 1728 cycles
Roundtrip IPIs on x64

CDF

95th percentile = 4640 cycles

Cycles measured from BSP (core 0)
Multicast IPIs on x64

95$^{\text{th}}$ percentile $\sigma = 2812$ cycles
Unicast IPI vs memory polling

CDF of cycles measured from BSP (core 0) for unicast IPI, synchronous event (mem polling), coherence network, and interrupt network.
Unicast IPI vs memory polling

CDF

- unicast IPI
- synchronous event (mem polling)
- projected remote syscall

Cycles measured from BSP (core 0)
Unicast IPI vs memory polling

CDF

Unicast IPI
Synchronous event (mem polling)
Projected remote syscall

Cost of dest. handling

Cycles measured from BSP (core 0)
Single wakeup on phi

- **pthread condvar**
  - $\mu = 25722.7$
  - $\sigma = 618.407$
  - min = 24333
  - max = 27834

- **futex wakeup**
  - $\mu = 15637.4$
  - $\sigma = 1173.37$
  - min = 13311
  - max = 22343

- **unicast IPI**
  - $\mu = 740.85$
  - $\sigma = 5.71205$
  - min = 732
  - max = 761
Wake up deviation on phi

CDF

pthread condvar
futex broadcast
broadcast IPI
Wakeup deviation on phi

HW lower bound

user-mode events

~79000x
Single wakeup on x64 (CDF)

CDF

Cycles to Wakeup

U. mode condvar
Futex wakeup
Oneway IPI
Single wakeup on phi (CDF)

CDF

Cycles to Wakeup

U. mode condvar
Futex wakeup
Oneway IPI
Single wakeup on phi

Cycles to Wakeup

- pthread condvar
  - min = 24333
  - max = 27834
  - μ = 25722.7
  - σ = 618.407

- futex wakeup
  - min = 5528
  - max = 14074
  - μ = 9014.73
  - σ = 2507.14

- Aerokernel condvar
  - min = 13311
  - max = 22343
  - μ = 15637.4
  - σ = 1173.37

- Aerokernel condvar + IPI
  - min = 5953
  - max = 7305
  - μ = 6483.89
  - σ = 213.517

- unicast IPI
  - min = 732
  - max = 761
  - μ = 740.85
  - σ = 5.71205
Many wakeups on phi

There are different ways to handle wakeups in a system. The diagram shows the number of cycles required to wake up different threads using various methods:

- **pthread condvar**: Min = 31743, Max = 1.17807e+07, μ = 5.64881e+06, σ = 3.08505e+06
- **futex wakeup**: Min = 28545, Max = 4.55009e+06, μ = 2.25096e+06, σ = 1.27551e+06
- **Aerokernel condvar**: Min = 7199, Max = 7.25596e+06, μ = 1.02075e+06, σ = 1.1411e+06
- **Aerokernel condvar + IPI**: Min = 8267, Max = 3.93142e+06, μ = 558155, σ = 428849
- **broadcast IPI**: Min = 1016, Max = 1225, μ = 1153.68, σ = 17.1824

The diagram visually represents the distribution of cycles required for each method, with box plots showing the minimum, maximum, and mean values.
Wakeup deviation on phi

![Graph showing CDF of wakeup deviation for different methods: pthread condvar, futex broadcast, Aerokernel condvar, Aerokernel condvar + IPI, broadcast IPI. The x-axis represents the deviation in seconds (\(\sigma\)), and the y-axis represents the cumulative distribution function (CDF).]
Wakeup deviation on phi

CDF

HW lower bound

user-mode

events

4x

Nemo events

(kernel-mode)

pthread condvar
futex broadcast
Aerokernel condvar
Aerokernel condvar + IPI
broadcast IPI

1000000
10000000
Broadcast wakeups on phi

- **pthread condvar**
  - $\mu = 5.64881 \times 10^6$
  - $\sigma = 3.08505 \times 10^6$
  - $\text{min} = 31743$
  - $\text{max} = 1.17807 \times 10^7$

- **futex wakeup**
  - $\mu = 2.25096 \times 10^6$
  - $\sigma = 1.27551 \times 10^6$
  - $\text{min} = 28545$
  - $\text{max} = 4.55009 \times 10^6$

- **broadcast IPI**
  - $\mu = 1153.68$
  - $\sigma = 17.1824$
  - $\text{min} = 1016$
  - $\text{max} = 1225$
Single wakeup on phi (CDF)
racket multiverse overheads

Runtime (s)

Native  
Virtual  
Multiverse

fannkuch-redux  
binary-tree-2  
fasta  
fasta-3  
nbody  
spectral-norm  
mandelbrot-2
system call overheads

~40µs