Resource Containers
A New Facility for Resource Management in Server Systems

Written by
Gaurav Banga, Peter Druschel @Rice University
Jeffrey C. Mogul @Western Research Laboratory, Compaq Computer Corp.

Presented in USENIX - OSDI'99
Motivation

No system API to control the resource consumption
- Hard to prevent DoS attacks
- Hard to provide a differentiated QoS for each Client

Problems
- Application are not able to control some resource which it consumes
- No proper resource management unit to control for application

Goals
- Separate Resource principal from Protection domain
- Provide proper API to control resources for application
Motivation

Protection domain
- A domain which used for one activity and it should be not used for others

Resource principal
- A unit for resource allocation and account

* A process is a protection domain and simultaneously a resource principal in general OS

Modern server applications

Resource management mechanisms of current OS
Typical models for high-performance servers

- Thread scheduler: time-sharing
- Thread: user-level or kernel
- Idle thread: for listening socket
- Ideal model in theory

Some problems from sustaining many connections with a small set of process

Fig. 3: A single-process multi-threaded server.
Shortcomings of current resource management models

Process in “Process-centric Systems”

Fairness
Among resource principals / under various load conditions
Chargeable
allocation of resources
Process
A schedulable entity
An unit which constitutes an independent activity
- Protection domain : isolation between applications
- Resource Principal : OS’s resource management subsystem

Several cases in which the natural boundaries of resource principals DO NOT coincide with either processes or threads
The distinction between scheduling entities and activities

Fig. 4: A classical application.

Fig. 5: A classical network-intensive application.
The distinction between scheduling entities and activities

Fig. 6: A multi-process application.

Fig. 7: A single-process multi-threaded server.
Integrating network processing with resource management

**Shortcomings**

- **LRP**
  - Lazy Receiver Processing: A Network Subsystem Architecture for Server Systems
  - Packet’s owner handles packets
    - -> more accurate accounting

- **Still, a process is a resource principal**

- **No client priority**

Fig. 8: A network-intensive application in a LRP system.
Resource Container: A new model for resource management

“Resource Container”
Abstract OS entity
  CPU time / Sockets / Protocol control blocks / Network buffers ...

Attributes
  Scheduling parameters / Resource limits / Network QoS ...
Access Control Model

Account
  Store consumed resources’ account in a container
Application can access and adjust priority
A new model for resource management
Resource Container: A new model for resource management

Scheduling
Desired allocation of CPU time, recent history of actual usage

Binding
Thread

Resource Container A
Resource Container B
Resource Container C
Resource Container D
Resource Container: A new model for resource management
Resource Container: A new model for resource management

Fig. 9: Containers in a multi-threaded server.

Fig. 10: Containers in an event-driven server.
Resource Container: A new model for resource management

sockaddr

- For Prioritization
- "filter"
  The way to distinguish request sources
- Binding 0-priority resource container to malicious client
- Provide better QoS for a premium client
- Enable precise accounting for the costs of an activity
Performance: Specification

Digital UNIX 4.0D Kernel
- CPU scheduler / resource management subsystem / network subsystem

Digital Personal Workstation 500au [Server]
- 500MHz 21164 / 128MB RAM
- A single-process event-driven program
  * [Client] 166MHz Pentium Pro / 64MB RAM / FreeBSD 2.2.5, S-Client SW

Private 100Mbps switched Fast Ethernet
Performance: Main Web Server Process

- Prioritized handling of clients

![Graph showing response time vs. number of concurrent low-priority clients with different configurations: Unmodified, select() with RC, Resource Container, and With containers/new event API. The y-axis represents response time in milliseconds, and the x-axis represents the number of concurrent low-priority clients. The graph compares the performance of different configurations over the range of clients.]
Performance: for Dynamic Resource

- Controlling resource usage of CGI processing
Performance: Prevent a kind of DoS attack

- Immunity against SYN-flooding
Conclusion

- Protection domain / Resource principal
  - Replace with Resource Container and thread
  - Hard to control resources in User-level

- Resource Container
  - Bound to thread which uses that resource
  - Provide an account information for scheduler
  - Its priority can be adjusted in user-level

- It is operated more effective in server with low latency
  - Provide more accurate account information and client filtering feature
Reviewer’s Opinion

Problem 1. Set a priority

Problem 2. Inaccurate account (for CPU usage)

CFQ scheduler
- CFQ I/O scheduler in LINUX 2.6.6 (2004. 5. 10)
- Became the default scheduler in LINUX 2.6.18 (2006. 9. 20)
- Adjust a nice value

http://superuser.com/questions/309063/how-can-i-prioritise-network-bandwidth-on-a-per-application-basis
Reviewer’s Opinion

Reviewer’s Opinion

Problem 3. Other resources and ...

- Only CPU is managed with time-sharing: Memory, FS, etc?
- How to set a priority for each client
Appendix
Typical models for high-performance servers

- Problem 1
  - Forking Overhead
    -> Pre-forking

- Problem 2
  - Context switching cost
  - IPC Overhead

Fig. 1: A process-per connection HTTP server with a master process.
Typical models for high-performance servers

- `select()` system call
  - Check all connection’s events
  - Handle them in a main loop

Fig. 2: A single-process event-driven server.
Resource Container: A new model for resource management

Operations
- Create / Release
  Created during fork()
  Bound with the first thread
  Exposed as file descriptor to application
- Share
  Pass a container between thread (like fd)
- Attributes
  It can be set and read by application
- Usage information
  Provide some resource information for application
- Binding
  Bind a container to thread at any time
- Reset
  Application can reset its own resource container
Performance

- Costs for new Primitives

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>create resource container</td>
<td>2.36</td>
</tr>
<tr>
<td>destroy resource container</td>
<td>2.10</td>
</tr>
<tr>
<td>change thread’s resource binding</td>
<td>1.04</td>
</tr>
<tr>
<td>obtain container resource usage</td>
<td>2.04</td>
</tr>
<tr>
<td>set/get container attributes</td>
<td>2.10</td>
</tr>
<tr>
<td>move container between processes</td>
<td>3.15</td>
</tr>
<tr>
<td>obtain handle for existing container</td>
<td>1.90</td>
</tr>
</tbody>
</table>

Table 1: Cost of resource container primitives.
Lazy Receiver Processing

Figure 1: BSD Architecture

Figure 2: LRP Architecture
SYN-flooding and Denial of Service

- Denial of Service
  특정 서버에게 수많은 접속 시도를 만들어 다른 이용자가 정상적으로 서비스 이용을 하지 못하게 하거나, 서버의 TCP 연결을 바닥내는 등의 공격이 이 범위에 포함

- SYN flooding attack
  DoS의 일종, TCP의 약점을 이용한 공격

- TCP 3 way handshaking
  1. A client -> B server로 접속 요청의 SYN packet 전송
  2. B는 A로 요청 수락의 SYN, ACK packet 전송
  3. A는 다시 B로 ACK를 보내고 실제 data 교환 시작

여기서 ACK를 기다리며 server는 Half Open 상태 일정 시간 후 server는 연결을 초기화 하지만, 그 전까지 backlog queue를 누적
-> 공격자는 SYN을 빠르게 보내 backlog queue를 overflow 시킴
Background about GPGPU* for understanding Ptask**

*General-Purpose computing on Graphics Processing Units
**PTask : Operating System Abstractions To Manage GPUs as Compute Devices
• High density computing resources
  - GTX 980 has 2048 CUDA cores and 4612GFLOPs (floating point operations per second).

• SIMT (Single Instruction Multiple Threads) execution model

Fig 1. GM204 Full-chip block diagram
GPU accelerated computing

• Programmers enable to write high-performance code for GPU hardware via GPGPU frameworks such as OpenCL, CUDA, and DirectX.

Fig 2. how GPU acceleration works
Example of parallel workloads

```c
void vector_add(double* a, double *b, double *c, size_t n)
{
    size_t i;
    for(i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}

__global__ void vector_add(double* a, double *b, double* c)
{
    tid = blockIdx.x*threadblocksize + threadIdx.x;
    c[tid] = a[tid] + b[tid];
}
```

Fig 3. serial process of vector addition

Fig 4. parallel process of vector addition (CUDA style)
GPGPU has been successful in specific domains

Fig 5. NVIDIA TESLA K80 accelerator performance
Despite the success of GPUs in super computing

GPGPU is not routinely integrated into many other types of system because

• Programming difficulty
• Lack of modularity
• Unpredictable performance artifacts
Difficult to programming

• Memory copy between CPU(host) and GPU(device)
  • Synchronous buffer, asynchronous buffer, CUDA stream and pinning of memory buffers

• To earn high performance, programmer have to understand about GPU architecture.
  • Memory access pattern, private memory, parallelism by using of resources, L1 cache memory and so on.
Lack of modularity

• \((AxB)xC = ?\)

Matrix gemm(A, B) {
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    invokeGPU(gemm_kernel, A, B, res);
    copyFromDevice(res);
    return res;
}

Fig 6. Pseudo-code of gemm’s subroutine
Modular $A \times B \times C$

Matrix $\text{modularAxBxC}(A, B, C)\{
  \text{matrix AxB = gemm(A, B);} \\
  \text{matrix AxBxC = gemm(AxB, C);} \\
  \text{return AxBxC;}
\}

Fig 7. Pseudo-code of modular $A \times B \times C$

Matrix gemm(A, B) \{
  \text{matrix res = new matrix();} \\
  \text{copyToDevice(A);} \\
  \text{copyToDevice(B);} \\
  \text{invokeGPU(gemm_kernel, A, B, res);} \\
  \text{copyFromDevice(res);} \\
  \text{return res;}
\}

Fig 6. Pseudo-code of gemm’s subroutine
Modular AxBxC

Matrix modularAxBxC(A, B, C){
    matrix AxB = gemm(A, B);
    matrix AxBxC = gemm(AxB, C);
    return AxBxC;
}

Matrix gemm(A, B) {
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    invokeGPU(gemm_kernel, A, B, res);
    copyFromDevice(res);
    return res;
}

Fig 7. Pseudo-code of modular AxBxC

Fig 6. Pseudo-code of gemm’s subroutine

Fig 8. GPU timeline of modular AxBxC
## Non-modular $\text{AxBxC}$

```java
Matrix nonmodularAxBxC(A, B, C)
{
    matrix intermd = new matrix();
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(gemm_kernel, A, B, intermd);
    invokeGPU(gemm_kernel, intermd, C, res);
    return res;
}
```

Fig 9. Pseudo-code of non-modular $\text{AxBxC}$
Non-modular AxBxC

Matrix nonmodularAxBxC(A, B, C){
    matrix intermd = new matrix();
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(gemm_kernel, A, B, intermd);
    invokeGPU(gemm_kernel, intermd, C, res);
    return res;
}

Fig 9. Pseudo-code of non-modular AxBxC

Fig 10. GPU timeline of non-modular AxBxC
PTask: Operating System Abstractions to Manage GPUs as Compute Devices

Written by C. J. Rossbach, et al.
Microsoft Research, Technion, University of Texas at Austin

Presented in SOSP’11
Motivation

• GPUs used in many applications
  • Gaming, HPC/batch
• GPU’s memory and main memory are disjoint
  • Unusable in other application domains
• Current OS treats GPUs as I/O devices

• New OS abstractions are needed for GPUs
OS-level abstractions

Traditional perspective

- No kernel-facing API
- No OS resource-management
- Poor composability

GPU’s perspective

- CUDA APIs
- Shaders/Kernels
- Language integration
OS Scheduling on CPUs and GPU

**GPU-bound processes hurt CPUs**
- OS cannot prioritize cursor updates
  - *WDDM + **DWM + CUDA == dysfunction

**CPU-bound processes hurt GPUs**
- The inability of Windows 7 to load balance a system

*Window Display Driver Model*  
**Desktop Window Manager**  
*Note: Average CPU utilization is under 25%*
GPU execution model

- GPUs cannot run OS: different ISA
- Disjoint memory space, no coherence
- Host CPU must manage GPU execution
  - Program inputs explicitly transferred/bound at runtime
  - Device buffers pre-allocated

User-mode program must implement to handle these
Composition: gestural interface

- High data rates
- Data-parallel algorithms: good fit for GPU!
What they aim to do

• Modular design
  • flexibility, reuse
• Utilize heterogeneous hardware
  • Data-parallel components – GPU
  • Sequential components – CPU
• Using OS provided tools
  • Processes, pipes

#> capture | xform | filter | detect &
  CPU  
  camera images | GPU  
  geometric transformation | GPU  
  noise filtering | CPU  
  detect gestures
Data migration

Double-buffering issue occurs!

user

capture

xform

filter

detect

read() write() read() write() read() write() read()

kernel

camdrv

GPU driver

HIDdrv

HW

GPU

Run!

*IRP

*I/O Request Packet used by WDM

OS

copy to GPU
copy from GPU

copy to GPU
copy from GPU

read()
write()
read()
write()
read()
GPUs need better OS abstractions

• The additional system calls analogous to
  • Process API
  • IPC API
  • Scheduler hints

• Abstractions that enable
  • Fairness/isolation
  • OS use of GPU
  • Composition/data movement optimization
**PTask design**

- New OS abstractions to support GPU programming
- PTask API = interfaces + runtime library support
- Support a dataflow programming model
  - Assemble individual tasks into a directed acyclic graph (DAG)
  - Vertices (ptasks) are executable code
    - Shader program on GPU
    - Code fragments on other accelerators
    - Callbacks on CPUs
  - Edges represent data flow
- To bring GPUs under the purview of a single resource manager
- To provide a simple programming model for accelerators
- To allow code to be both modular and fast
PTask OS abstractions (1)

- PTask
  - Priority for fairness
  - Analogous to a process for GPU execution
  - List of input/output resources (e.g. stdin, stdout, stderr ...)

- Ports
  - Mapped to ptask input/outputs
  - A data source or sink
PTask OS abstractions (2)

• Channels
  • Similar to pipes, connect arbitrary ports
  • Specialize to eliminated double-buffering

• Graph
  • Direct Acyclic Graph connected ptasks, ports, channels
  • The unit that PTask runtime will schedule fairly
PTask OS abstractions (3)

- Datablocks
  - Memory-space transparent buffers
  - Logical buffer
    - Backed by multiple physical buffers
    - Buffers created/updated lazily
    - Mem-mapping used to share across process boundaries
  - Track buffer validity per memory space
    - Writes invalidate other views
  - Flags for access control/data placement

<table>
<thead>
<tr>
<th>space</th>
<th>V</th>
<th>M</th>
<th>RW</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>gpu0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>gpu1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

...
# PTask API system calls

<table>
<thead>
<tr>
<th>PTask system call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_open_graph</td>
<td>Create/open a graph</td>
</tr>
<tr>
<td>sys_open_port</td>
<td>Create/open a port</td>
</tr>
<tr>
<td>sys_open_ptask</td>
<td>Create/open a ptask</td>
</tr>
<tr>
<td>sys_open_channel</td>
<td>Create and bind a channel</td>
</tr>
<tr>
<td>sys_open_template</td>
<td>Create/open a template</td>
</tr>
<tr>
<td>sys_push</td>
<td>Read from a channel/port</td>
</tr>
<tr>
<td>sys_run_graph</td>
<td>Run a graph</td>
</tr>
<tr>
<td>sys_terminate_graph</td>
<td>Terminate graph</td>
</tr>
<tr>
<td>sys_set_ptask_prio</td>
<td>set ptask priority</td>
</tr>
<tr>
<td>sys_set_geometry</td>
<td>Set iteration space</td>
</tr>
</tbody>
</table>
**PTask scheduling modes**

- **First-available**
  - Every ptask is assigned to a manager thread
  - Threads competing for available accelerators (GPUs)
  - Arbitrated by locks on accelerators
  - Not queued

- **FIFO**
  - first-available + queue

- **Priority**
  - PTask enhanced with a static priority and proxy priority
  - Normalized to OS priority
  - User-settable priority
  - Sorts the queue based on each ptask’s priority

- **Data-aware**
  - Priority, but assigns GPUs for input locality
More about priority mode

- **Static priority**
- **Proxy priority**
  - OS priority of the thread managing its invocation and data flows
  - To avoid *proxy laundering*
  - Enable a ptask’s manager to assume the priority of a requesting process

---

**OS**

```
7
requesting
```

**Ptask’s manager**

```
2
```

**Proxy laundering**

```
7
```

**With proxy priority**

```
7
```
PTask Scheduler

• Manages a ready queue of ptasks and a list of available accelerators
• Computes an effective priority value for ptasks on wake-up
  • Effective priority = ptask’s static priority + boost values
  • Boost values derived from
    • its current wait time, its average wait time,
      its average run time, its proxy priority
  • Avoids starvation
  • Increases throughput
  • Respects the priority of OS process scheduler
• Assigns an accelerator to a ptask based on fitness and strength
  • Fitness = support of the execution environment and feature for a ptask
  • Strength = # of cores * core clock speed * memory clock speed
    • Imperfect but effective heuristic for ranking accelerators
      such that low-latency execution is preferred
PTask invocation

- Waiting (for inputs)
- Queued (inputs available, waiting for a GPU)
- Executing (running on the GPU)
- Completed (finished execution, waiting to have its outputs consumed)
PTask graph: gesture interface

- Optimized data movement
- Data arrival triggers computation
Datablock action: gesture interface

Datablock

<table>
<thead>
<tr>
<th>space</th>
<th>V</th>
<th>M</th>
<th>RW</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gpu</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Main Memory

GPU Memory

- process
- ptask
- port
- channel
- datablock
Revised technology stack

- GPU APIs can be built on top of new OS abstractions
Implementation

- Windows 7
  - Full PTask API implementation
  - Stacked UMDF/KMDF driver
    - Kernel component: mem-mapping, signaling
    - User component: wraps DirectX, CUDA, OpenGL
  - syscalls: DeviceIoControl() calls
- Linux 2.6.33.2
  - PTask not implemented
    - Implement only GPU scheduling, a non-work-conserving scheduling
    - Similar to the token bucket algorithm
  - Changed OS scheduling to manage GPU
    - GPU accounting added to `task_struct`
GPU scheduling on Linux

- Using the GPU can defeat the kernel’s scheduling priority
  - The one invoking longer GPU kernels -> monopolization
- Necessary to implement a new scheduling on Linux
  - Add GPU accounting state to Linux’s `task_struct`
  - Add blocking system calls informing the kernel about GPU activity
  - Each process $p$ using GPU
    - budget $B_p$: current eligibility to use the GPU
    - $B_p$ reduced by the execution time used by $p$ each execution on a GPU
    - $B_p$ increased per period $T$ by quanta $q$
    - $B_p < 0$: the kernel will block GPU invocations until $B_p > 0$
    - $B_{max} = \frac{n_p}{\sum_{i\in P} n_i} \sum_{i\in P} t_i$, $q = T = \alpha \times \min_{i\in P} t_i$
  - $P$: the set of kernel threads using a GPU
  - Updated upon every GPU completion, invocation or completion of a process using a GPU, and every 10 usec (1/10 lower than GPU kernel time)
Evaluation on Linux (1)

- Core i5 3.20GHz, 12GB RAM, GTX470, 2*SATA SSD 80GB
- Run EncFS, a FUSE-based encrypted file system for Linux
  - Modified to use a GPU for AES encryption and decryption
- Sequential read and write of 200MB file
  - Below data is relative to the CPU performing encryption
- GPU use can invert scheduling priority
  - if Linux is not informed about GPU use
  - leading to a drastic degradation of performance (30x slower)

<table>
<thead>
<tr>
<th></th>
<th>1 GPU task</th>
<th>2 GPU tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>Linux</td>
</tr>
<tr>
<td>Read</td>
<td>247MB/s</td>
<td>-10.0x</td>
</tr>
<tr>
<td>Write</td>
<td>82MB/s</td>
<td>-8.2x</td>
</tr>
</tbody>
</table>
Evaluation on Linux (2)

- Necessary to maintain global scheduling priorities
- Two EncFS threads of different nice values
  - Doing read or write concurrently on two CPUs
  - Contending for the GPU
- Without the new GPU scheduling (PTSched)
  - Throughput is not affected by the nice level

<table>
<thead>
<tr>
<th></th>
<th>r/w (nice)</th>
<th>Linux</th>
<th>PTSched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (0)</td>
<td>170</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>Read (-20)</td>
<td>171</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>Write (0)</td>
<td>58</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>Write (-20)</td>
<td>58</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>
Gesture interface evaluation

- Windows 7, Core2-Quad, GTX580 (EVGA)
- Implementations
  - pipes: capture | xform | filter | detect
  - modular: capture+xform+filter+detect, 1 process
  - handcode: data movement optimized, 1 process
  - ptask: ptask graph
- Configurations
  - real-time: driven by cameras
  - unconstrained: driven by in-memory playback
Gesture interface performance

![Graph showing Gesture interface performance](graph.png)

- **runtime**: Handcode, Modular, Pipes, Ptask
- **user**: Handcode, Modular, Pipes, Ptask
- **sys**: Handcode, Modular, Pipes, Ptask
Multi-GPU scheduling on Windows

- GPU-GPU data migration
  - priority: 14.6%, data-aware: 0.6%

- Data-aware $==$ priority + locality
Performance isolation on Windows

- Concurrently run 4 PTask graphs on a single GPU
  - Each of 36 ptasks
Conclusions

• OS abstractions for GPUs are critical
  • Enable fairness & priority
  • OS can use the GPU
• Dataflow: a good fit abstraction
  • System manages data movement
  • Performance benefits significant
Reviewer’s opinions

• Cannot support problems that are not well-expressed as static graphs
  • Based on dataflow programming model (DAG)
• Still requires CPUs to invoke PTask
• Feasible to work with new technologies?
  • NVLink, GPUDirect RDMA, and so on
• Windows and Linux have different OS designs
  • How to ensure it will work on other OSes?
  • Not fully implemented on both of the OSes
    • Even for Windows, just the Camera driver modified
  • Possible to make it run for other purposes?
  • Not clearly understood
    • Lack of full explanation
    • Closed source