KRASH
CPU Load Generation on Many-Core Machines

Guillaume Huard and **Swann Perarnau**

Moais INRIA Team, CNRS LIG Lab, Grenoble University
France

Tuesday, 20 April 2010

Outline

Evolution of computing machines

Towards Many-Core Systems

No more CPU frequency scaling.
Who has the biggest number of cores ?

Simple system used as computing servers

- Single machine with 4-32 cores
- No use as a desktop
- Used as shared computing servers

Towards Heterogeneous Systems

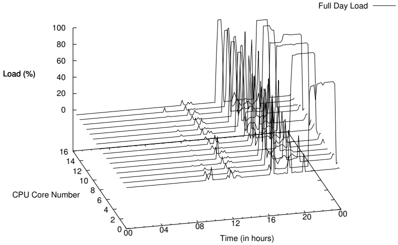
Shared machines

- Multiple users, multiple programs in a single address space
- Fair Scheduling Policy

Specialized cores

- GPU inside CPU
- Variable frequencies
- Hybrid architectures (ex: Cell)

Dynamic Heterogeneity



Shared Memory, Many-Core systems present a **dynamic heterogeneity**.

Classical Parallel Programming Challenged

New Paradigms for Heterogenous Systems

- fine grained parallelization + work-stealing [Blumofe 1995]
- adaptive parallel algorithms [Roch & al. 2008]

New Evaluation Criteria

- Struggling capacity for resources access (out of our scope)
- Efficient use of given resources

How to Compare Them ?

We must use a controlled environment.

Our Goal: Help experimental evaluation

Classical scientific methodology

- Identify a representative environment,
- Compare algorithms in this same experimental conditions.

Challenge : design a load generation tool

- Producing heterogeneous environment from dedicated machines.
- Guarantying reproducibility.
- Insensitive to other applications efforts to take control of resources.

In this first work, we focus on CPU load only

Outline

Timeslicing and Resolution

Timeslice

Period of exclusive access to a CPU core.

CPU Load (during time interval)

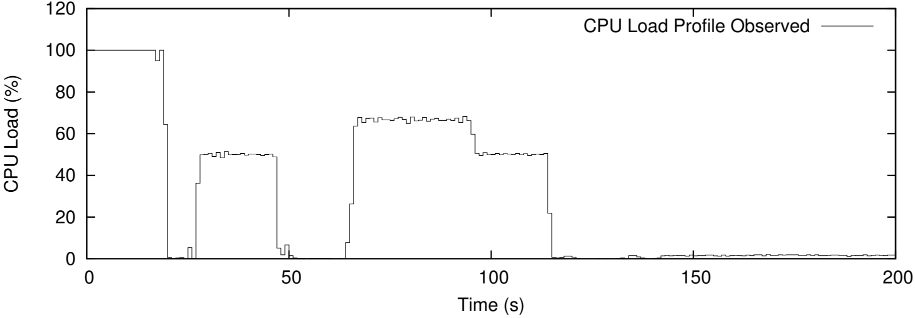
Proportion of unavailability : core assigned to some process or in the kernel

Timeslice Size

Quite large : typically 1ms to 10ms

- Fair share between processes is only reached asymptotically.
- Notion of **scheduler resolution**.

Example Load Profile



Load profile of a dynamic environment
(several NAS instances started and stoped at various times) run in concurrence with a Linpack instance

A Good Load Generator

Requirements

- Precision, Reactivity
- Noise insensitivity

Must Avoid

- Intrusiveness
- Obtrusiveness

Outline

Various methods

Run a CPU intensive process. $\Rightarrow$ No control on the load.

Various methods

Run a CPU intensive process.	$\Rightarrow$	No control on the load.
Adjust priority during execution.	$\Rightarrow$	Load sensitive to environment.

Various methods

Run a CPU intensive process.	$\Rightarrow$	No control on the load.
Adjust priority during execution.	$\Rightarrow$	Load sensitive to environment.
Realtime process.	$\Rightarrow$	Changes scheduler behavior.

Various methods

Run a CPU intensive process.	$\Rightarrow$	No control on the load.
Adjust priority during execution.	$\Rightarrow$	Load sensitive to environment.
Realtime process.	$\Rightarrow$	Changes scheduler behavior.
Wrekavoc: Start/Stop applications [Jeannot,2009].	$\Rightarrow$	Scales badly, hiders scheduling heuristics.

Various methods

Run a CPU intensive process.	⇒	No control on the load.
Adjust priority during execution.	⇒	Load sensitive to environment.
Realtime process.	⇒	Changes scheduler behavior.
Wrekavoc: Start/Stop applications [Jeannot,2009].	⇒	Scales badly, hiders scheduling heuristics.
Scale CPU frequency.	⇒	Modifies hardware charactiristics.

Existing Solutions Issues

50% load during NAS DT (communications intensive MPI application).

	Execution time			
	avg	std. dev.	Slowdown	Issue
None	2.9	0.5	1	-
Real time priority	11.3	3.2	3.9	less comm./comp. overlapping
Start/Stop	NA	NA	> 100	Supervision overhead
Frequency scaling	4.4	0.6	1.5	more comm./comp. overlapping

Current methods fail to load complex applications.

Outline

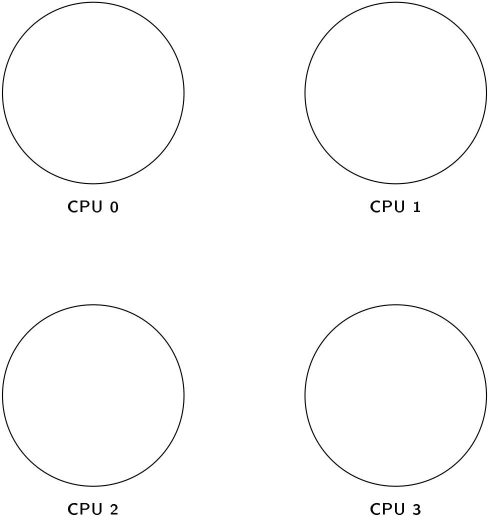
Our claim

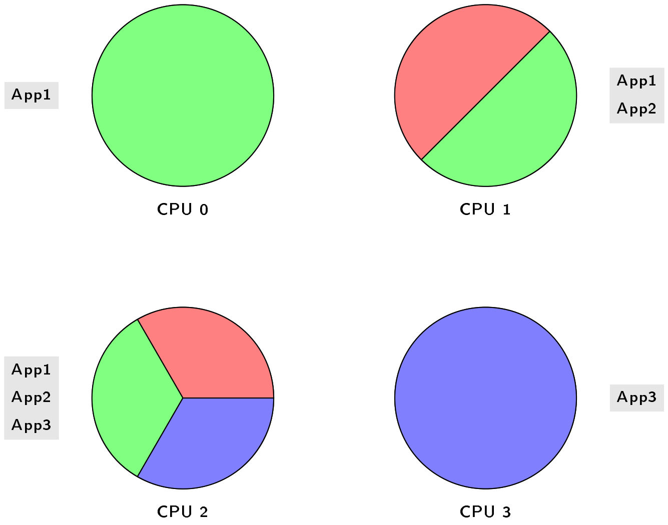
Interact directly with the Scheduler

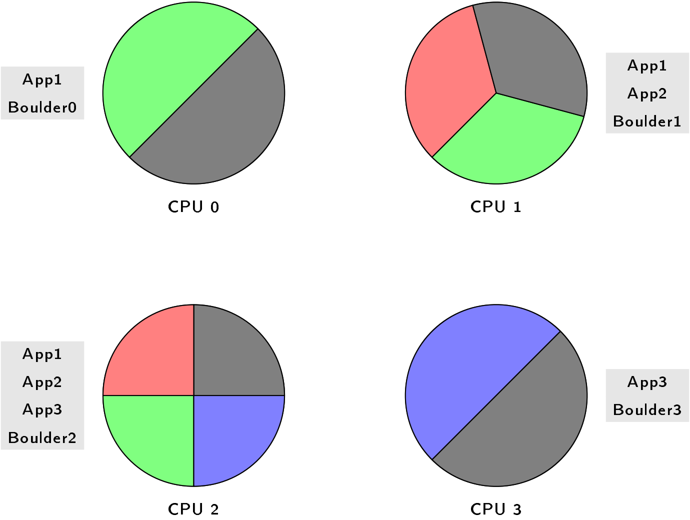
- Precision, Reactivity ensured.
- Noise insensitive.
- (Very) Low Intrusiveness.
- No Obtrusiveness.

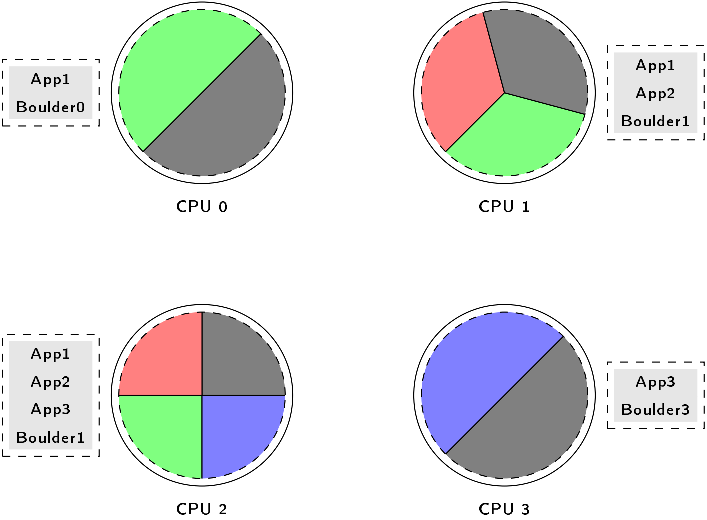
Our Load Generation Method

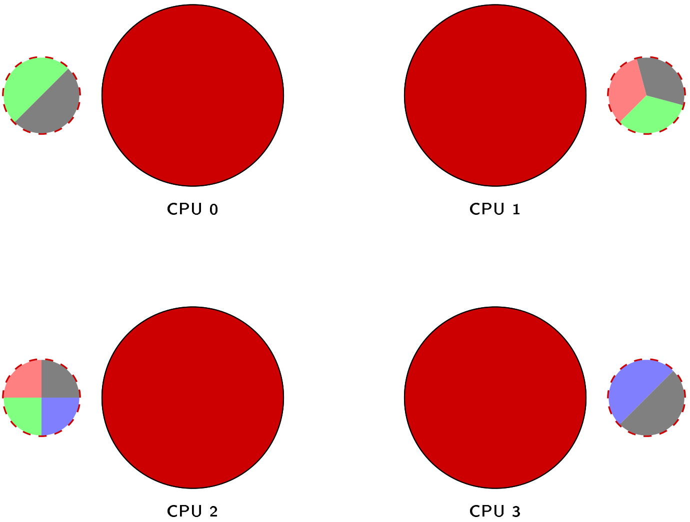
1. Use **cpu affinity** to attach a process (**boulder**)to each core.
2. Use **group scheduling** to assign timeslices to boulders.
3. use a **supervisor** process to adjust group priorities when needed.

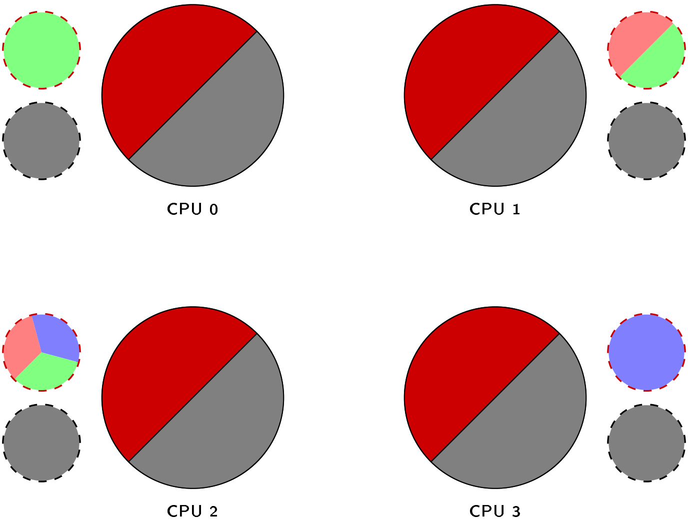

CPU 0
CPU 1
CPU 2
CPU 3

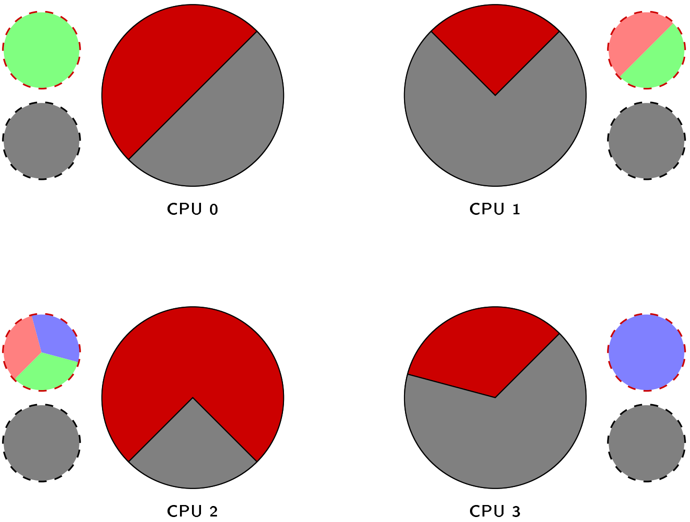

App1
CPU 0
App1
App2
CPU 1
App1
App2
App3
CPU 2
App3
CPU 3


App1
Boulder0
CPU 0
App1
App2
Boulder1
CPU 1
App1
App2
App3
Boulder2
CPU 2
App3
Boulder3
CPU 3


App1
Boulder0
CPU 0
App1
App2
Boulder1
CPU 1
App1
App2
App3
Boulder1
CPU 2
App3
Boulder3
CPU 3


CPU 0
CPU 1
CPU 2
CPU 3


CPU 0
CPU 1
CPU 2
CPU 3


CPU 0
CPU 1
CPU 2
CPU 3

KRASH

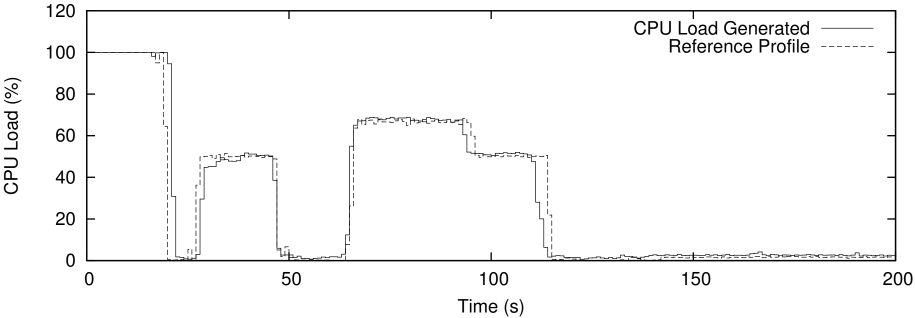
C/C++ Implementation

- Linux only,
- Supervising process mostly sleeping,
- Using **control cgroups** VFS.

Input

```
cpu {                        # begin cpuinj
    ...                      # config params
    profile {                # the load profile itself
        0 {                  # cpuid to load
            0    70          # load 70% of the cpu
            60    30         # after one minute only 30% the cpu
            120    50
        }
        1 {                  # another cpu loaded
            0    70
            10    30
        }
    }                        # end of profile
}                            # end cpuinj
kill 150                     # stop krash after 150 seconds
```

KRASH Load Reproduction



Reproduction performed by KRASH run in concurrence with a Linpack instance

Outline

In the Paper

Qualitative Comparison

- Dynamic load profile,
- Modification of the scheduler,
- Maximum number of loaded process,
- Resolution

Quantitative Comparison

- Precision and Intrusiveness
- Effects on I/O
- Effects on Network
- Effects on complex applications

In the Paper

Qualitative Comparison

- Dynamic load profile,
- Modification of the scheduler,
- Maximum number of loaded process,
- Resolution

Quantitative Comparison

- Precision and Intrusiveness
- Effects on I/O
- Effects on Network
- **Effects on complex applications**

Quantitative comparison

KRASH is the only tool able to generate dynamic load profiles
$\Rightarrow$ comparison will be limited to constant load generation

Test platform : NUMA SMP machine

- 8 dual-core Opteron 875
- 32 GB RAM
- 250 GB Raid 1 storage subsystem

Obtrusiveness: Mixed (tasks duration, I/Os, CPU)

Constant 50% load during parallel gcc compilation
(varying processes duration, mixed I/Os and CPU intensive tasks)

	Execution time average	Execution time standard deviation	Slowdown	Issue
None	197	3	1	-
KRASH	387	7	2	None
Wrekavoc	NA	NA	> 100	Too many processes $\Rightarrow$ Supervision overhead
Real time priority	558	5	2.8	RT FIFO prevents I/Os priority
Frequency scaling	392	21	2	None

I/Os less important than in case I, per process overhead critical

Outline

Conclusion and Future Works

Krash: a new CPU load generation method

- Behaves as would do a CPU intensive application,
- Reproducible load : precise, reactive and insensitive to noise,
- Unobtrusive : do not induces unexpected performance impact on other system. resources,

Future Works: extend Krash

- To cache (application cache trashing),
- To memory (bandwidth sharing),
- To I/Os (bandwidth/latency perturbations).

Thank you for your attention.

Krash is publicly available at `http://krash.ligforge.imag.fr/`