Improving the performance of data servers on multicore architectures

Fabien Gaud

Grenoble University

Advisors:
Jean-Bernard Stefani, Renaud Lachaize and Vivien Quéma

Sardes (INRIA/LIG)

December 2, 2010

Processor evolution

- Before $\sim$2006:
  - One core
  - Regular increase of clock frequency
- Since then:
  - Almost no increase of clock frequency
  - Increasing number of cores:
    - Multicore architectures
    - NUMA architectures
    - Manycore architectures

Multicore is a hot topic

- Legacy applications do not efficiently leverage multicore hardware
- Research topics:
  - Programming models/languages
  - Operating systems abstractions/internals
  - Runtime/libraries
  - Applications
- Active research field:
  - Corey (OSDI'08)
  - Barrelfish (SOSP'09), Helios (SOSP'09)
  - PK (OSDI'10)

This thesis

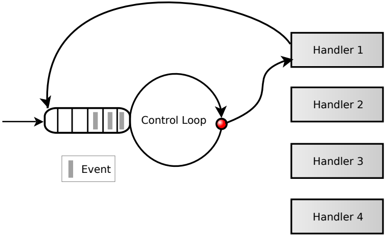
- **Application domain:** data servers, *a.k.a. networked services*
- **Goal:** Improve the performance of data servers on multicore architectures
- **Contributions:**
  - Efficient multicore event-driven programming
  - Scaling the Apache Web server on NUMA multicore systems

#1: Efficient multicore event-driven programming

CFSE 2009 (*best paper award*)
ICDCS 2010

Event-driven programming

- Application is structured as a set of **handlers** processing **events**
- An event can be:
  - Triggered by an I/O operation
  - Produced internally by the application
- Events are stored in a **queue** and processed by a **single thread**

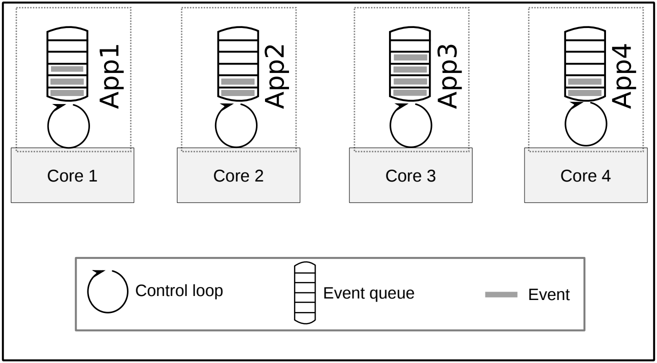


Multicore event-driven programming

- **Goal:** concurrently execute multiple handlers
- **Challenges:**
  - Concurrency management
  - Balancing load on cores
- **Solutions:**
  - N-Copy
  - 1-Copy with synchronization

N-Copy

- **Principle:** running one instance of the application per core



N-Copy (2)

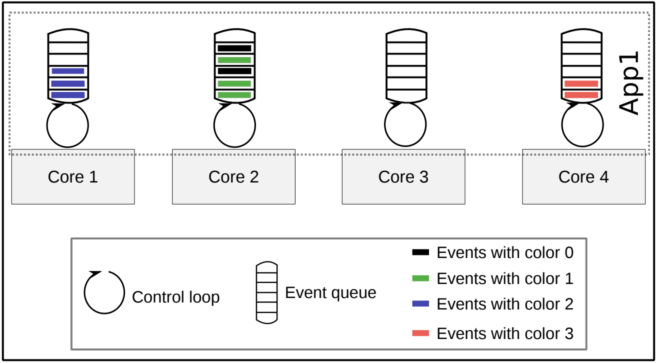
- **Advantages:**
  - No concurrency management needed
  - No application modification needed
- **Drawbacks:**
  - Not applicable to all applications
  - Multiple copies of data
  - Requires external load balancing

1-copy with synchronization

- **Principle:** 1 instance on multiple cores
- Concurrency can be managed using:
  - Locks
  - STM
  - Annotations
- Load balancing can be achieved with:
  - Static placement
  - Workgiving
  - Workstealing
- Chosen approach is implemented in Libasync-SMP (Usenix'03)

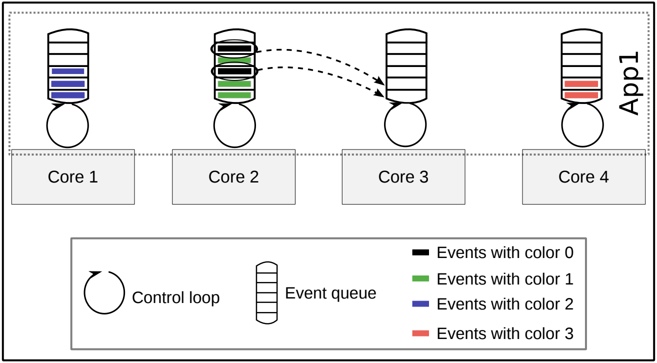
Libasync-SMP – Concurrency management

- **Annotations** *(colors)* set on events



Libasync-SMP – Load balancing

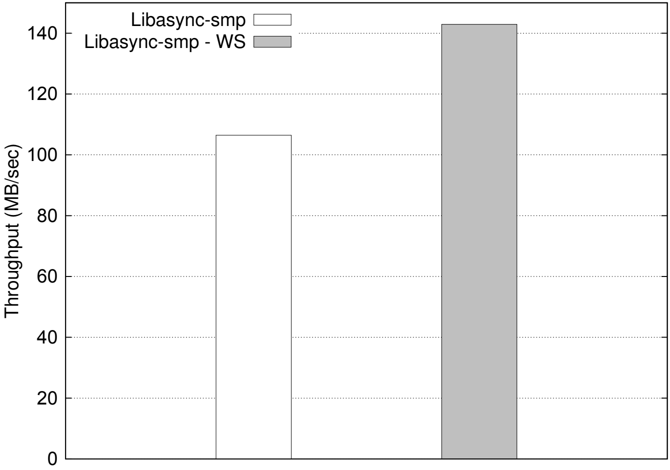
- Load balancing is done through **workstealing**



1-Copy with synchronization

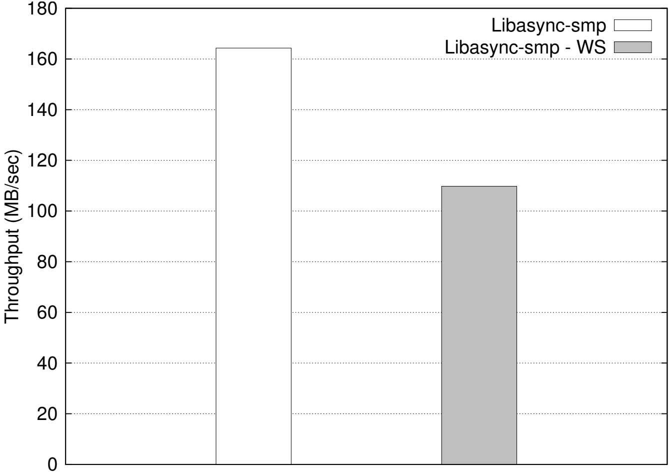
- **Advantages:**
  - Allows sharing between cores
  - Allows load balancing between cores
- **Drawbacks:**
  - Need to modify the application
  - Efficient load balancing is difficult

Workstealing performance: SFS



35% throughput increase

Workstealing performance: Web server



33% throughput decrease

What is the problem?

- Fine grain events:
  - Stealing time (197 Kcycles) $\gg$ stolen processing time (20 Kcycles)
- Inefficient cache usage:
  - +146% L2 cache misses
- Inefficient workstealing implementation
  - *O(n)* complexity

Contributions

- **New:**
  - Workstealing algorithm
  - Runtime implementation
- Fine grain events:
  - Algorithm: steal events with high execution time
- Inefficient cache usage:
  - Algorithm: steal cache-friendly events
  - Algorithm: take cache hierarchy into account
- Inefficient workstealing implementation
  - Runtime: mitigate stealing costs

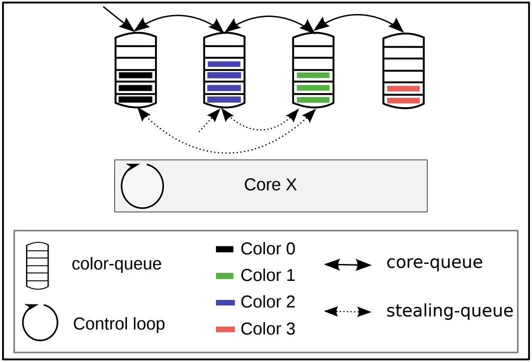
Idea #1: Take into account execution time

- **Problem:** stealing cost is not always amortized
  - Many event handlers are relatively fine grain
  - Workstealing may have a significant cost
- **Solution:** Time-left stealing
  - Know at any time which colors are *worthy*
  - (Handler execution time is set by the programmer)

Idea #2: Take into account caches

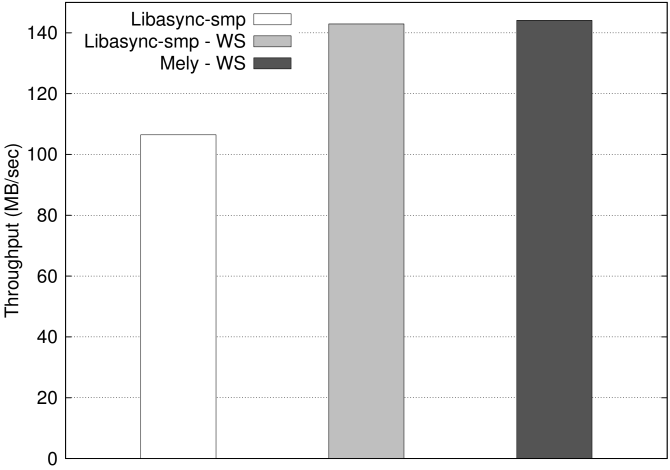
- **Problem:** Workstealing can reduce cache efficiency
  - Stealing events increases cache misses
  - Example: event handlers accessing large, long-lived, data sets
- **Solution 1:** Penalty-aware stealing
  - Set penalties on handlers based on their cache access pattern
  - (Penalties are set manually based on preliminary profiling)
- **Solution 2:** Locality-aware stealing
  - Give priority to a neighbor when stealing

Runtime implementation



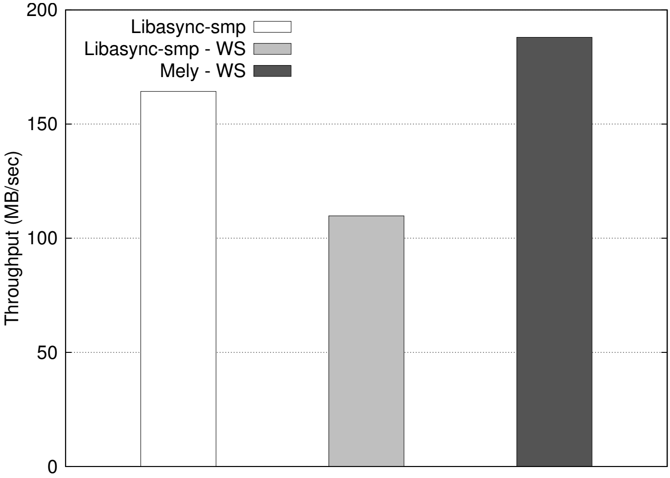
- One `color-queue` per color
- One `core-queue` per core that links `color-queues`
- One `stealing-queue` per core

Performance evaluation: SFS



No throughput degradation

Performance evaluation: Web server



73% throughput improvement

Web server profiling

Web server configuration	Stealing time	Stolen time	Cache misses/event
Libasync-SMP - WS	197 Kcycles	20 Kcycles	21
Mely - WS	6 Kcycles	23 Kcycles	9

- Stealing time (6 Kcycles) < stolen processing time (23 Kcycles)
- Improved cache efficiency: -57% L2 cache misses

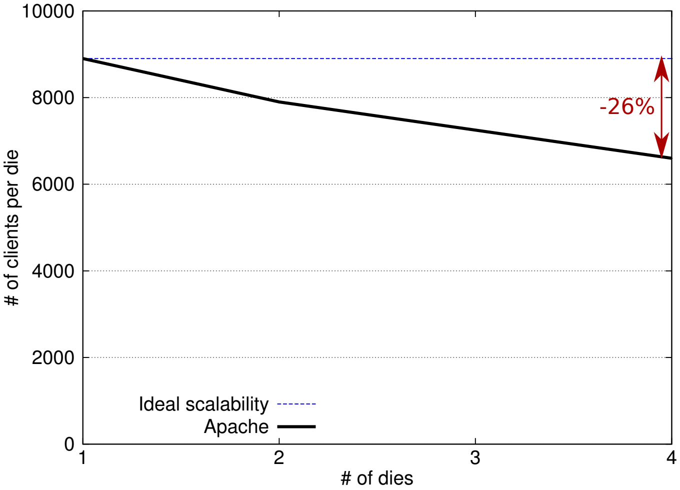
Summary

- **Goal:** efficient runtime for multicore event-driven systems
- **Problem:** workstealing sometimes degrades performance
- **Contributions:**
  - New workstealing algorithm
  - New runtime implementation
- **Results:** improve throughput by up to 73%

#2: Scaling the Apache Web server on NUMA multicore systems

Under submission

Problem

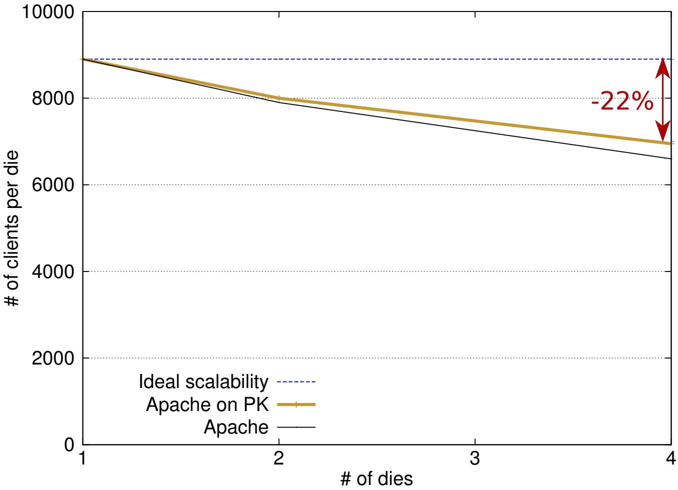


The Apache web server do not scale on NUMA architectures

What can we do?

- Address scalability issues at the OS level
  - Corey (OSDI 08)
  - Barrelfish (SOSP 09)
  - PK (OSDI 10)

Apache on PK



Does not solve scalability issues

What do we propose?

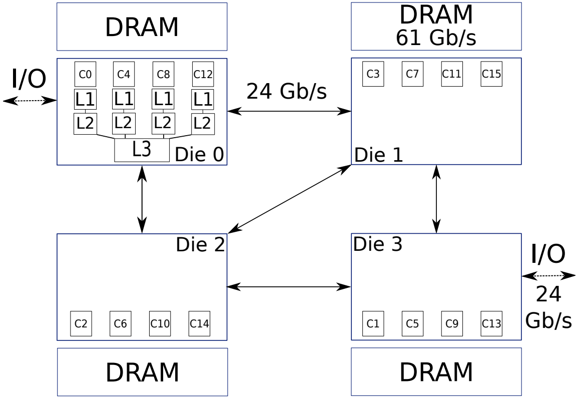
- Addressing scalability issues at the OS level is not sufficient
  - Application-level issues
  - Some issues are difficult to handle (e.g. scheduling)

- **Approach:** address scalability issues at the application level

Methodology

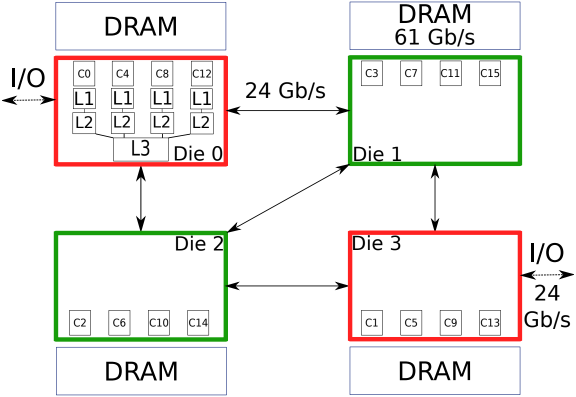
- Consider both hardware and software bottlenecks
- Hardware bottlenecks:
  - Processor interconnect
  - Distant memory accesses
- Software bottlenecks:
  - Synchronization primitives

Hardware testbed



- 4 processors / 16 cores

Hardware testbed



- 4 processors / 16 cores

Hardware bottlenecks

- Memory efficiency (IPC)

Configuration	Average IPC
1 die	0.38
4 dies	0.30

21% IPC decrease

Hardware bottlenecks (2)

- IPC decrease:
  - Reduced cache efficiency

Configuration	L3 cache miss ratio (%)
1 die	14
4 dies	14

Hardware bottlenecks (2)

- IPC decrease:
  - ~~Reduced cache efficiency~~
  - HyperTransport link saturation

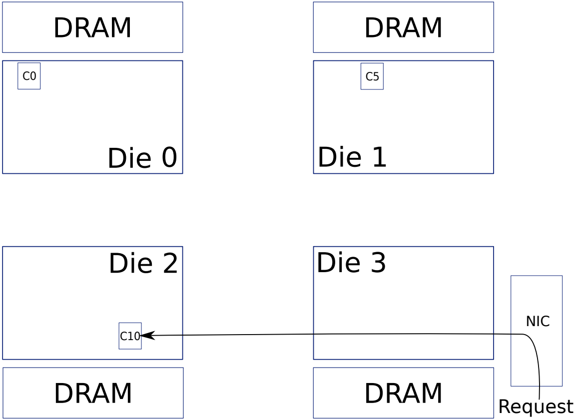
Configuration	Max HT usage (%)
1 die	25
4 dies	75

Hardware bottlenecks (2)

- IPC decrease:
  - ~~Reduced cache efficiency~~
  - ~~HyperTransport link saturation~~
  - Increased number of distant memory accesses

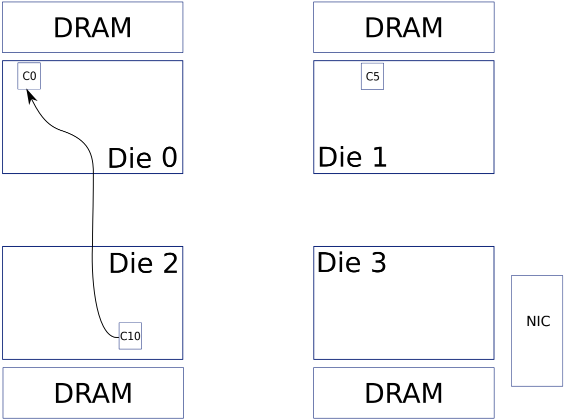
Configuration	**Distant accesses/kB**
1 die	4
4 dies	14

Request processing



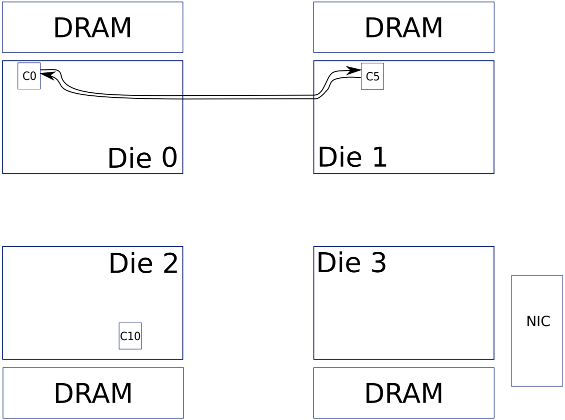
Receiving a TCP request

Request processing



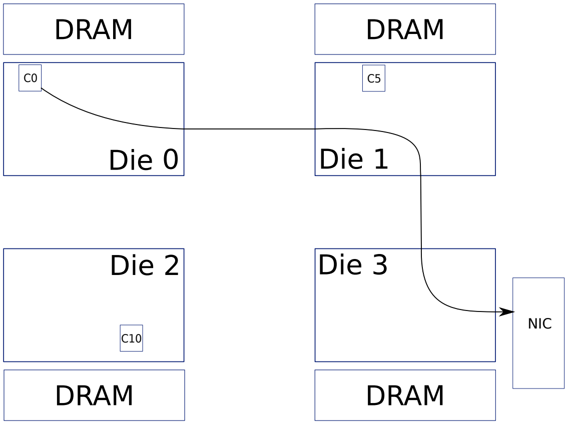
HTTP request processing

Request processing



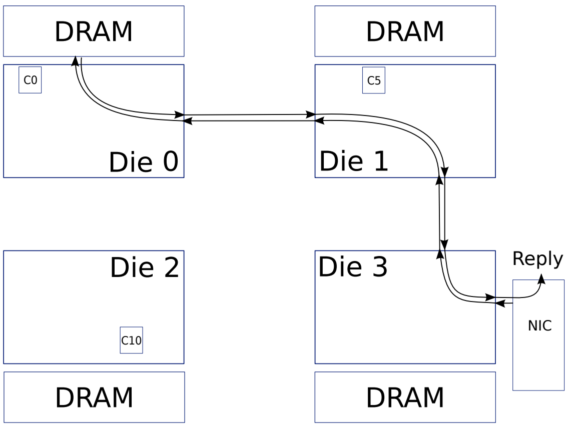
PHP processing

Request processing



Sending the response (1)

Request processing



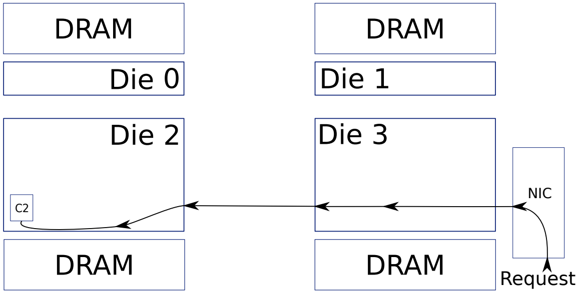
Sending the response (2)

Proposal #1

- **Solution:** co-localizing TCP, Apache and PHP processing

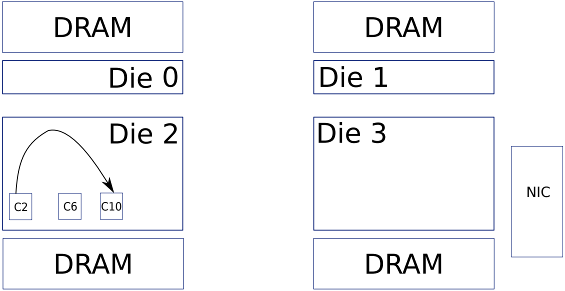
- **Implementation:** use one instance of the Apache/PHP stack per die *(N-Copy)*
  - One node manages 5 network interfaces

N-Copy: request processing



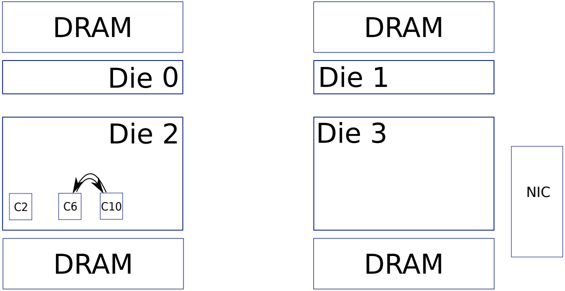
Receiving a TCP request

N-Copy: request processing



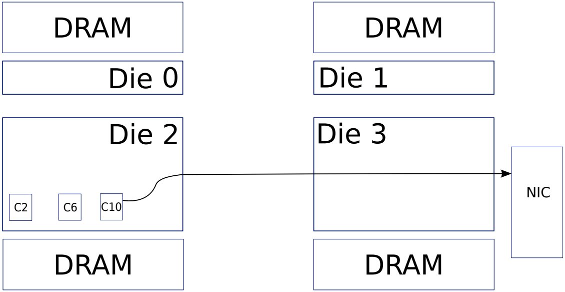
HTTP request processing

N-Copy: request processing



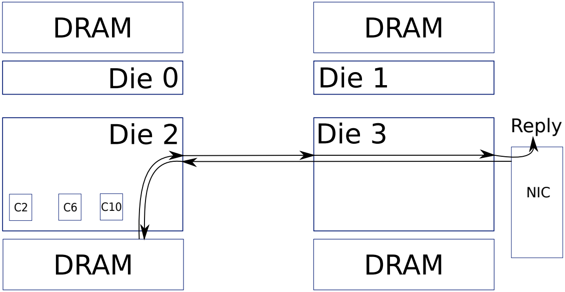
PHP processing

N-Copy: request processing



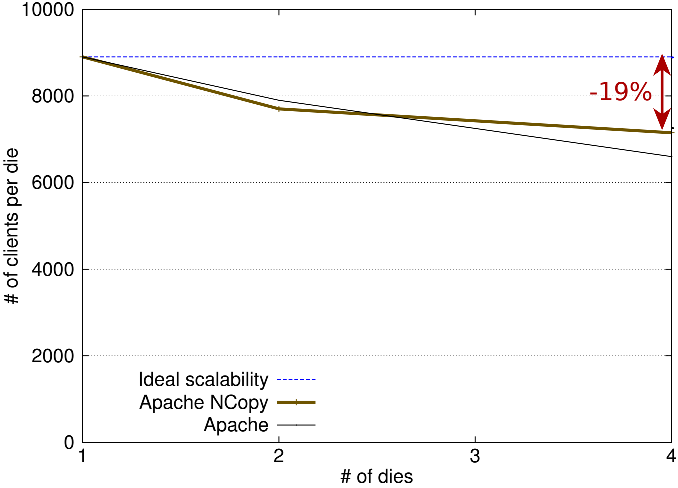
Sending the response (1)

N-Copy: request processing



Sending the response (2)

N-Copy: performance



9.1% performance improvement compared to stock Apache

N-Copy: performance (2)

Configuration	Average IPC	Distant accesses/kB
1 die	0.38	4
4 dies (Stock Apache)	0.30	14
4 dies (N-Copy)	0.36	5

Memory efficiency improved by 20%

N-Copy: can we do better?

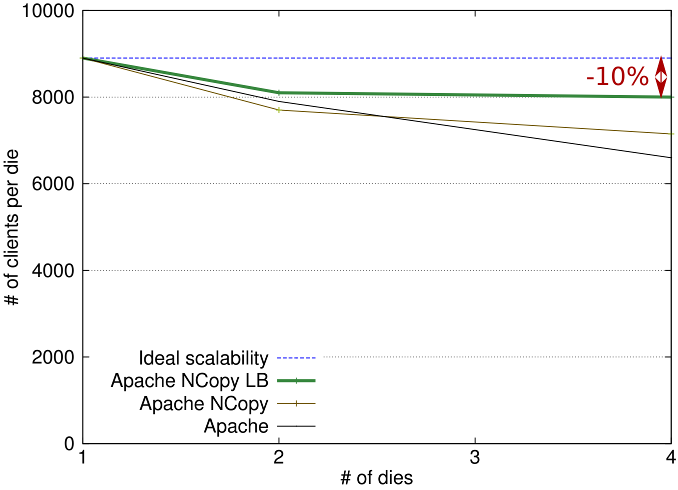
Die	Average CPU usage
Die 0	100
Die 1	85
Die 2	85
Die 3	100

- **Problem:**
  - Dies are not equally efficient
  - Load is not *properly* balanced on dies

N-Copy: load balancing

- **Solution:** balance load on dies proportionally to their efficiency
- **Implementation:** use an external load balancing mechanism
  - Currently implemented at client-side
  - Could be integrated in a more global solution

N-Copy: final performance



21.2% performance improvement compared to stock Apache

Software bottlenecks

- **Goal:** find functions that
  - Do not scale
  - Represent a significant execution time

- Example:
  - Function *f* accounts for
    - 1 cycle/byte at 1 die
    - 10 cycles/byte at 4 dies
    - 20% of the total execution time
  - 18% *potential performance gain*

Software bottlenecks (2)

Function	Potential performance gain (%)
__d_lookup	2.49%
_atomic_dec_and_lock	2.32%
lookup_mnt	1.41%
copy_user_generic_string	0.83%
memcpy	0.76%

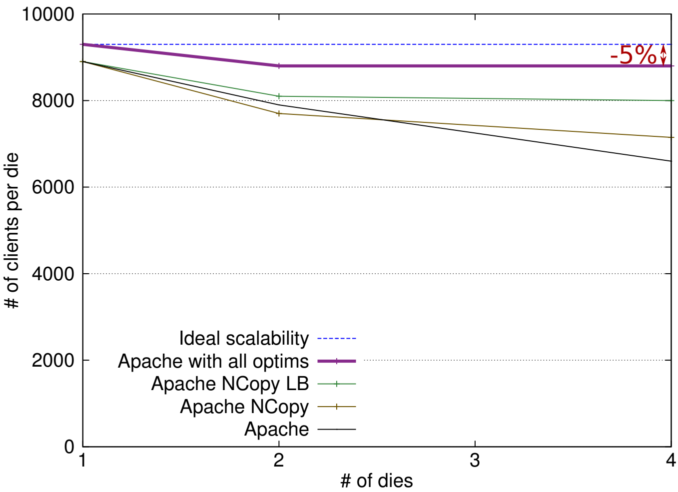
- **Problem:** the VFS layer does not scale
    - Aggregated potential performance gain: 6 %
    - Most of the calls are issued by the `stat` function

Proposal #2

- **Solution:** use an application-level cache to reduce the number of calls to `stat`

- **Implementation:**
  - Modified the Apache `ap_directory_walk` function
  - Using `inotify` for file updates

Stat cache: performance



33% performance improvement compared to stock Apache

Summary

- **Problem:** Apache does not scale on NUMA architectures
- **Contribution:** application-level optimizations considering NUMA aspects and Linux scalability issues
- **Results:** +33% performance improvement

Conclusion

Conclusion

- **Application domain:** data servers
- **Goal:** Improve the performance of data servers on multicore architectures
- **Contributions:**
  - Efficient multicore event-driven programming
  - Scaling the Apache Web server on NUMA multicore systems

Future work

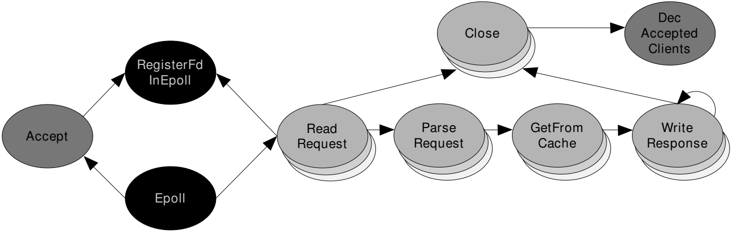
- Short term:
  - Workstealing: automate profiling and decisions
  - Apache: study other workloads
- Long term:
  - Study the impact of distant memory accesses on other servers
  - Study the impact of programming models on multicore performance
  - Study the scalability of the Java virtual machine

Questions?

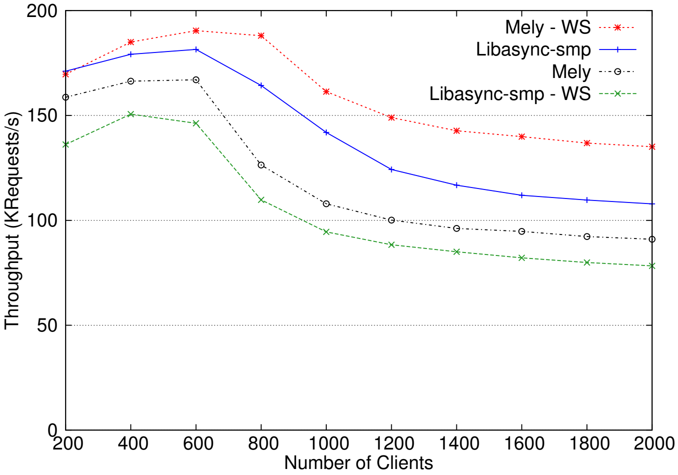
Backup Slides

Web server

- Returns static page content (1KB files requested)
- Closed-loop injection
- 5 load injectors simulating between 200 and 2000 clients
- Architecture is based on legacy design
  - Per-connection coloring

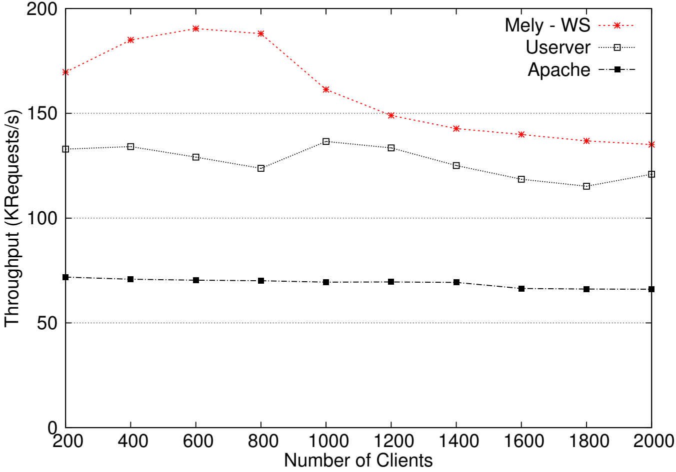


Web server evaluation



⇒ Up to 73% improvement over the Libasync-SMP workstealing mechanism

Mely - Other web server evaluation (2)



⇒ Performance better than other real world Web servers

Apache – Workload description

- SPECWeb2005 Support benchmark
  - Vendor site
  - Mostly static / PHP for dynamic pages
  - Back-end Simulator (BeSim)
- Closed-loop injection with think times
- Defined QoS:
  - 99% of clients served within 5s
  - 95% of clients served within 3s
  - Throughput constraints
- Modified to fit in main memory: 12GB

Software configuration

- Apache 2.2.14
  - *Worker* version using both threads and processes
  - `Sendfile` enabled to improve performance
- PHP 5.2.12
  - Tuned number of PHP processes
  - With eAccelerator
- Linux 2.6.32
  - NUMA support
  - IRQ processing manually balanced
  - Responsible for dispatching thread and processes