Efficient Workstealing for Multicore Event-Driven Systems

Fabien Gaud[1], Sylvain Genevès[1], Renaud Lachaize[1], Baptiste Lepers[2], Fabien Mottet[2], Gilles Muller[2], Vivien Quéma[3]

[1]University of Grenoble

[2]INRIA

[3]CNRS

International Conference on Distributed Computing Systems 2010

Outline

Objectives

- Application domain : data servers
- Focus on event-driven programming

- Multicore architectures are mainstream
- Exploiting the available hardware parallelism becomes crucial for data server performance

⇒ Our goal is to provide an efficient multicore runtime for event-driven programming

Event-driven runtime basics

- Application is structured as a set of *handlers* processing *events*.
- An event can be triggered by an I/O or produced internally
- The runtime engine repeatedly processes events from its queue
  - Get an event from the runtime's queue
  - Call the associated handler which may produce new events

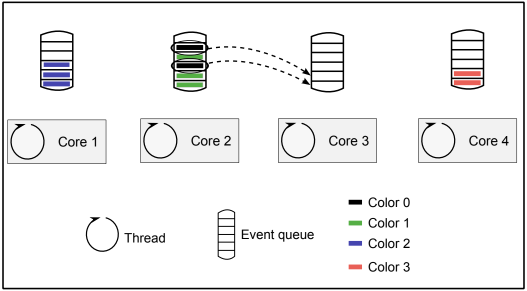
Multicore event-driven runtime

- Challenges
  - Helping programmers dealing with concurrency
    - Locks
    - STM
    - **Annotations**
  - Efficiently dispatching events on cores
    - Static placement
    - Load balancing through workgiving
    - **Load balancing through workstealing**

⇒ Libasync-SMP is an annotation-based multicore event-driven runtime

Libasync-SMP [Zeldovich03]

- One event queue per core
- Mutual exclusion ensured by annotations on events *(colors)*
- Event dispatching on cores
  - Colors are initially dispatched in a round robin manner
  - Load balancing is readjusted through workstealing

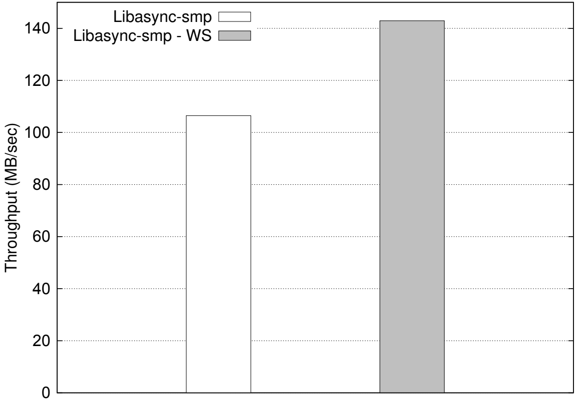


- Evaluation on two network servers
  - Workstealing is only evaluated on micro-benchmarks

Outline

Expected behavior : the SFS case

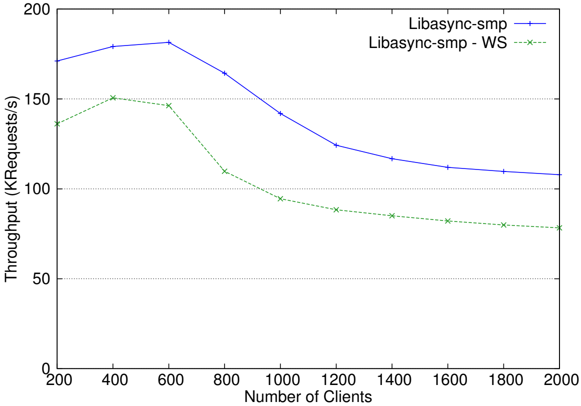
- Many expensive cryptographic operations
- Good case for workstealing algorithm
- Example : clients accessing a 200MB file



⇒ 35% throughput increase thanks to workstealing

Unwanted behavior : the Web server case

- Web server serving static content
- Workstealing costs are noticeable
- Example : clients accessing 1KB files



⇒ 33% throughput decrease due to the workstealing mechanism

Unwanted behavior : the Web server case (2)

Web server configuration	Stealing time	Stolen time	Cache misses / event
Libasync-SMP without workstealing	-	-	9
Libasync-SMP with workstealing	197 Kcycles	20 Kcycles	21

- Very high stealing costs $\gg$ stolen computing time
- Very low cache efficiency : +146% L2 cache misses over Libasync-smp without workstealing

Problem statement

- Naive workstealing can hurt system performance
- This paper improves workstealing performance for multicore event-driven runtimes
- Majors differences with workstealing for thread-based runtimes
  - Tasks are more fine grained
    - Sensitivity to stealing costs
  - One core can post tasks to another core
    - Cannot use efficient DEqueue structures [Chase05]
  - Stealing is constrained by colors
    - *O(n)* workstealing algorithm

Workstealing main steps

```
core_set = construct_core_set();                                    (1)
foreach(core c in core_set) {
    LOCK(c);
    if(can_be_stolen(c)) {                                          (2)
        color = choose_colors_to_steal(c);                          (3)
        event_set = construct_event_set(c, color);
    }
    UNLOCK(c);
    if(!is_empty(event_set)) {
        LOCK(myself);
        migrate(event_set);
        UNLOCK(myself);
        exit;
    }
}
```

Outline

Idea #1 : Taking hardware topology into account

```
core_set = construct_core_set();                    (1)
```

- In a multicore system, some cores usually share caches
- Time needed to access cached data is significantly faster than accessing them in main memory

- Idea : Take the cache hierarchy into consideration when stealing
- Locality-aware stealing ⇒ Give priority to a neighbor when stealing

Idea #2 : Taking into account computation length

```
if(can_be_stolen(c)) {                                  (2)
```

- Many event handlers are relatively fine grain
- In our context, workstealing may have a significant cost
- Idea : Stealing some type of events is not beneficial
- Time-left stealing : know at any time which colors are *worthy*
- Handler execution time is currently set by the programmer but could be discovered at runtime

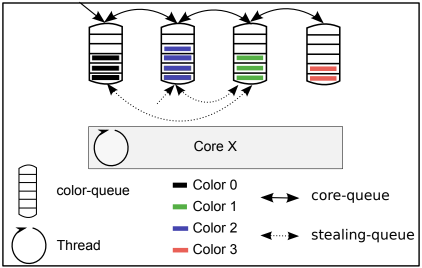
Idea #3 : Taking cache footprint into consideration

```
color = choose_colors_to_steal(c);          (3)
```

- Sometime events can be stolen but are not the best candidates
  - For example, event handlers accessing large, long-lived, data sets
- Penalty-aware stealing : giving penalty to events handlers based on their behavior
- Penalties are set by the programmer based on preliminary profiling and/or using application behavior knowledge

Outline

The Mely runtime

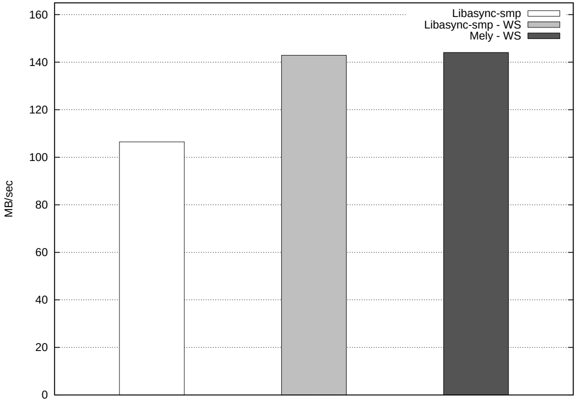


- Backward compatible with Libasync-SMP
- One thread per core
- One `color-queue` per color
- One `core-queue` per core that links `color-queues`
- One `stealing-queue` per core that allows to efficiently implement *Time-left* and *Penalty-aware* stealing strategies

Outline

SFS

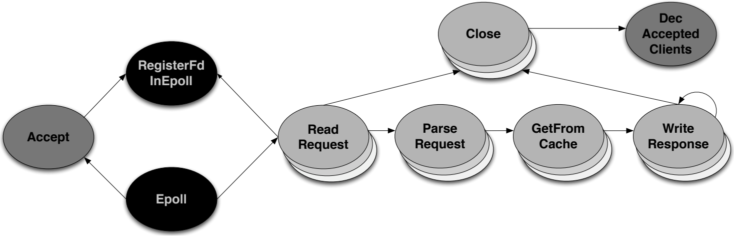
- 15 clients repeatedly request a 200MB file
- 60% time spent in cryptographic operations ⇒ only color cryptographic operations



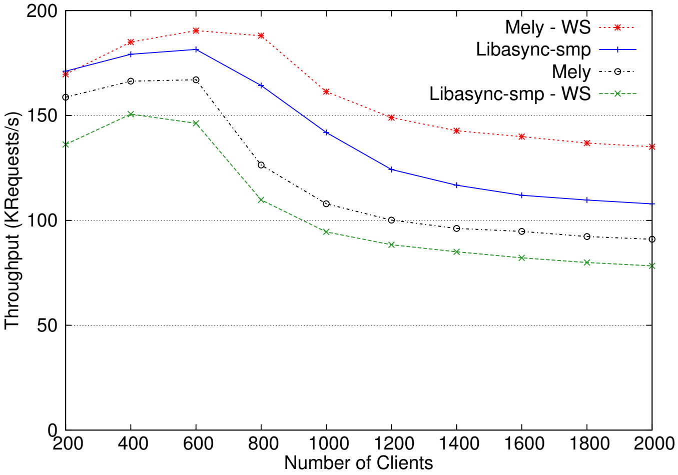
⇒ as expected same throughput as the legacy workstealing mechanism

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

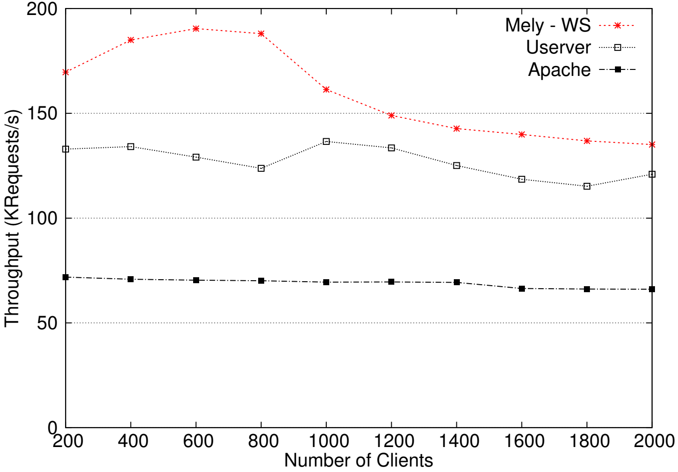


Web server evaluation



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

Web server evaluation (2)



⇒ Performances better than other real world Web servers

Web server profiling

Web server configuration	Stealing time	Stolen time	Cache misses / event
Libasync-SMP without workstealing	-	-	9
Libasync-SMP with workstealing	197 Kcycles	20 Kcycles	21
Mely with workstealing	6 Kcycles	23 Kcycles	9

- Low stealing overhead : 6 Kcycles $<$ stolen computing time
- Much more cache-efficient than Libasync-SMP
  - Locality and penalty aware heuristics decrease the number of L2 cache misses by 24%

Outline

Conclusion

- Context
  - Event driven programming for system services on multicore architectures
  - Workstealing sometimes degrades performances in such systems
- Contributions
  - New heuristics to improve workstealing efficiency
  - Revised runtime internals to reduce workstealing costs

  $\Rightarrow$ Improved Web server performance by 73% compared to the legacy workstealing mechanism.
- Future work : Automating runtime profiling and decision

Thank You !

Questions ?