Efficient Workstealing for Multicore Event-Driven Systems

Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, Vivien Quema

Efficient Workstealing for Multicore Event-Driven Systems

Fabien Gaud, Sylvain Genevès, Renaud Lachaize
University of Grenoble
France
first.last@inria.fr

Baptiste Lepers, Fabien Mottet, Gilles Muller
INRIA
France
first.last@inria.fr

Vivien Quéma
CNRS
France
vivien.quema@inria.fr

***Abstract*—Many high-performance communicating systems are designed using the event-driven paradigm. As multicore platforms are now pervasive, it becomes crucial for such systems to take advantage of the available hardware parallelism. *Event-coloring* is a promising approach in this regard. First, it allows programmers to simply and progressively inject support for the safe, parallel execution of multiple event handlers through the use of annotations. Second, it relies on a workstealing algorithm to dynamically balance the execution of event handlers on the available cores.**

This paper studies the impact of the workstealing algorithm on the overall system performance. We first show that the only existing workstealing algorithm designed for event-coloring runtimes is not always efficient: for instance, it causes a 33% performance degradation on a Web server. We then introduce several enhancements to improve the workstealing behavior. An evaluation using both microbenchmarks and real applications, a Web server and the Secure File Server (SFS), shows that our system consistently outperforms a state-of-the-art runtime (Libasync-smp), with or without workstealing. In particular, our new workstealing improves performance by up to +25% compared to Libasync-smp without workstealing and by up to +73% compared to the Libasync-smp workstealing algorithm, in the Web server case.

***Keywords*-workstealing; multicore; system services; performance; event-driven;**

I. Introduction

Event-driven programming is a popular approach for the development of robust applications such as networked systems [1], [2], [3], [4], [5], [6], [7], [8]. This programming and execution model is based on *continuation-passing* between short-lived and *cooperatively-scheduled* tasks. Its strength mainly lies in its expressiveness for fine-grain management of overlapping tasks, including asynchronous network and disk I/O. Moreover, some applications developed using the event-driven model exhibit lower memory consumption and better performance than their equivalents based on threaded models [9], [10].

However, a traditional event-driven runtime cannot take advantage of the current multicore platforms since it relies on a single thread executing the main processing loop. To overcome this restriction, a promising approach, *event coloring*, has been proposed and implemented within the Libasync-smp library [11]. Event coloring tries to preserve the serial event execution model and allows programmers to incrementally inject support for safe parallel execution through annotations (*colors*) specifying events that can be handled in parallel. The main benefits of the *event coloring* approach are that it preserves the expressiveness of pure event-driven programming, offers a relatively simple model with respect to concurrency, and is easily applicable to existing event-driven applications.

A side-effect of *event coloring* is that it sometimes causes unbalances in the processing load handled by the different cores of a machine. To improve performance, Libasync-smp designers have thus proposed a workstealing (WS) mechanism in charge of balancing event executions on the multiple cores. We actually show in this paper that enabling workstealing can hurt the throughput of real systems services by as much as 33%. Using microbenchmarks, we have identified two reasons for this performance decrease. First, the workstealing mechanism makes naïve decisions. Second, data structures used in the runtime are not optimized for workstealing.

The contributions of this paper are twofold. First, we introduce enhanced heuristics to guide workstealing decisions. These heuristics try to preserve cache locality and avoid unfavorable stealing attempts, with little involvement required from the application programmers. We then present Mely (*Multi-core Event LibrarY*), a novel event-driven runtime for multicore platforms. Mely is backward-compatible with Libasync-smp and its internal architecture has been designed with workstealing in mind. Consequently, Mely exhibits a very low workstealing overhead, which makes it more efficient for short-running events.

We evaluate Mely with a set of micro-benchmarks and two applications: a Web server and the Secure File Server (SFS) [12]. Our evaluations show that Mely consistently outperforms (or, at worse, equals) Libasync-smp. For instance, we show that the Web server running on top of Mely achieves a +25% higher throughput than when running on top of Libasync-smp without workstealing, and a +73% higher throughput than when running on top of Libasync-smp with workstealing enabled.

The paper is structured as follows. We start with an analysis of Libasync-smp in Section II. We then propose new heuristics to improve event workstealing in Section III. The implementation of the Mely runtime is presented in Section IV. Section V is dedicated to the performance evaluation of Mely. Finally, we discuss related work in Section VI, before concluding the

paper in Section VII.

II. The Libasync-smp runtime

This section describes the Libasync-smp runtime [11]. We start with a description of its design. Then, we detail the workstealing algorithm used to dynamically balance events on cores. Finally, we evaluate and analyze Libasync-smp performance on two real-sized system services.

A. Design

Libasync-smp is a multiprocessor-compliant event-driven runtime. Its implementation relies, for each core, on an event queue and a thread. Events are data structures containing a pointer to a handler function, and a *continuation* (i.e. a set of parameters carrying state information). Event handlers are executed by the core thread associated with the event queue. Handlers are assumed to be non-blocking, which explains why only one thread per core is required. The architecture of the Libasync-smp runtime is illustrated in Figure 1.

Since several threads (one per core) are simultaneously manipulating events, it is necessary to properly handle the concurrent execution of different handlers. An event execution updating a data item must execute in mutual exclusion with other events accessing the same data item. To ensure this property, Libasync-smp does not rely on the use of locking primitives in the code of the handlers. Rather, mutual exclusion issues are solved at the runtime level using programmer specifications. More precisely, programmers can restrain the potential parallel execution of events using annotations (named *colors* and represented as a short integer). Two events with different colors can be handled concurrently, whereas events of the same color must be handled serially. This is achieved by dispatching those events on the same core. Note that, events without annotations are all mapped to a default unique color in order to guarantee safe execution. The Libasync-smp implementation assigns new events to cores using a simple hashing function on colors. Load balancing is adjusted with a workstealing algorithm described in Section II-B.

Interestingly, the coloring algorithm allows implementing various forms of parallelism. For instance, it is possible to let multiple events associated to the same handler run concurrently on disjoint data sets (e.g., to ensure that different client connections are concurrently processed in a Web server). It is also possible to enforce that all events associated to the same handler be executed in mutual exclusion (e.g., when a handler manages global state).

Event coloring is less expressive than locking (e.g. it does not support reader-writer semantics) but less error-prone and sufficient for the needs of most server applications. Besides, events can still be combined with locks in the rare cases where mutual exclusion must span several handlers.

Because event queues can be concurrently updated by different cores, their access must be synchronized. This is implemented using spinlocks; indeed, there is no interest in yielding cores (only one thread per core), if energy is not a concern.

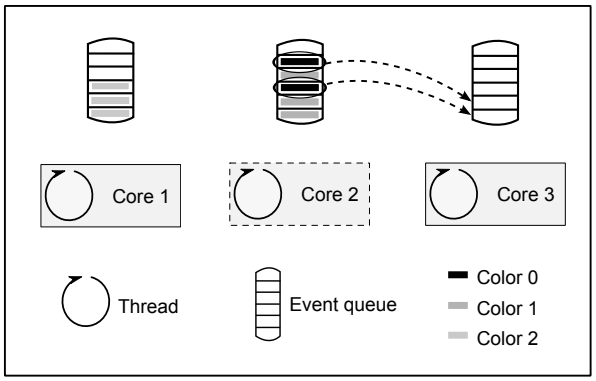

Figure 1. Libasync-smp architecture.

B. Workstealing algorithm

As mentioned in the previous section, colored events are dispatched on the cores using a *hashing* function. This simple load balancing strategy ignores the fact that some colors might require more time to be processed than others (e.g., when there are many events with the same color or when different events have different processing costs). The Libasync-smp library thus provides a dynamic load balancing algorithm based on the workstealing principle. When a core has no more events to process, it attempts to fetch events from other core queues.

The workstealing algorithm is presented as pseudo-code in Figure 2. First, the stealing core builds a `core_set` containing an ordered set of cores. This is achieved calling the `construct_core_set` function (functions used in the pseudo-code are detailed in the next paragraph). For each core in the set, the stealing core checks whether events can be stolen using the `can_be_stolen` function. If events can be stolen from this core, the stealing core chooses one color to be stolen using the `choose_color_to_steal` function. The stealing core then builds a set containing all the events with the chosen color using the `construct_event_set` function. If this set is not empty, the stealing core migrates the set of events in its own queue using the `migrate` function.

We now describe the implementation of the above mentioned functions. `construct_core_set` builds a set that contains as first element the core that has the highest number of events in its queue. The set then contains the successive cores (based on core numbers): for instance, on a 8-core computer, if core 6 currently contains the highest number of events, then `core_set` is equal to {6, 7, 0, 1, 2, 3, 4, 5}. The call to `can_be_stolen` returns true if the core given as parameter has at least events with two different colors in its queue. Indeed, two colors are required because, in order to enforce the mutual exclusion properties of the runtime, the color of the event currently being processed on a core cannot be stolen. A steal can thus only be performed if there are events with another color. `choose_color_to_steal` scans the event queue of the core given in parameter and selects the first color (i) that is not associated with the event currently being processed, and (ii) that is associated with less than half

of the events in the queue. Note that such a color might not exist. The `construct_event_set` function builds a set comprising all events stored in the queue of the stolen core that are associated with the color given as parameter. Moreover, it also removes events from the victim queue. Note that this function might require scanning the entire event queue. This is the case when the last event stored in the queue has the color given as parameter[1]. Finally, the `migrate` function appends a set of events to the queue of the stealing core.

```
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);          (4)
   }
   UNLOCK(c);
   if(!is_empty(event_set)) {
      LOCK(myself);
      migrate(event_set);                                 (5)
      UNLOCK(myself);
      exit;
   }
}
```

Figure 2. Pseudo code of Libasync-smp workstealing algorithm.

C. Performance evaluation

Zeldovich et al. have evaluated the performance of the Libasync-smp library on two system services: the SFS file server [12] and a Web server, which is not publicly available. While this study shows that the bare Libasync-smp achieves speedups on multicore platforms, workstealing has not been fully evaluated[2].

Therefore, we have developed a realistic Web server based on the design described in [11], and we have run both SFS and our Web server with workstealing enabled and disabled. Details on the Web server and the benchmark configuration (hardware and software settings) can be found in Section V. For all experiments, standard deviations are very low (less than 1%).

Figure 3 shows the throughput achieved by SFS when 16 clients are issuing read requests on a 200MB file. It highlights that the workstealing algorithm significantly improves the server throughput (+35%). The reason is that it mostly executes expensive, coarse-grain cryptographic operations.

In contrast, Figure 4 shows the throughput of the Web server with a varying number of clients requesting 1KB files. It clearly shows that the performance is negatively impacted by the workstealing algorithm (up to -33%). The reason is that the Web server relies on shorter event handlers than the ones used in SFS. Consequently, workstealing costs are proportionally higher.

To better understand the previous results, we measured the average time spent to steal a set of events (for both SFS and the Web server) and the average time spent to execute this set of stolen events. Results are summarized in Table I. We observe that the time spent to perform a steal (impacting one or several events) in SFS is on average 4.8 Kcycles and allows stealing sets of events whose average processing time is 1200 Kcycles. In contrast, a steal in the Web server requires a drastically longer average time (197 Kcycles) and allows stealing sets of events whose average processing time is much shorter (20 Kcycles).

[1]However, this is not always necessary since the runtime maintains a counter of pending events for each color.

[2]More precisely, the initial publication on Libasync-smp has only studied the impact of workstealing on a microbenchmark.

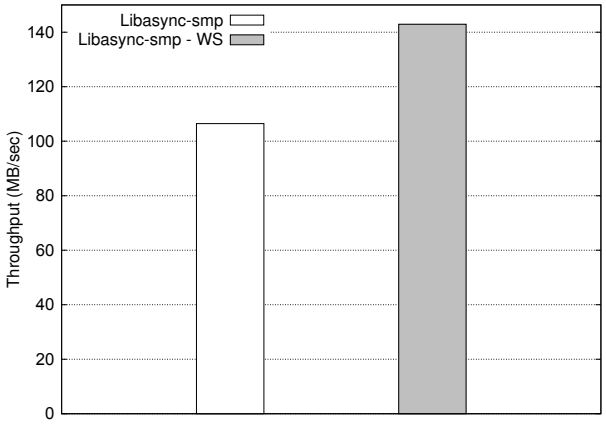

Figure 3. Performance of the SFS file server with and without workstealing algorithm.

We attribute the poor performance achieved by the Web server when workstealing is enabled to two main causes. First, the Libasync-smp workstealing algorithm is naïve: a stealing core never checks the relevance of a steal before performing it. More precisely, the `construct_core_set`, `can_be_stolen` and `choose_color_to_steal` functions do not take into account the cost of the steal, nor the processing time of the stolen events.

Table I
TIME SPENT STEALING A SET OF EVENTS VS. TIME SPENT EXECUTING THESE EVENTS.

System	Stealing time (cycles)	Stolen time (cycles)
SFS	4.8K	1200K
Web server	197K	20K

Moreover, the `construct_core_set` function does not consider cache proximity between cores. We monitored the number of L2 cache misses on the Web server and we observed a large increase of up to +146% when enabling workstealing. This result suggests that an efficient workstealing algorithm should try to favor dispatching events on cores sharing a L2 cache.

Second, the implementation of Libasync-smp has not been designed with workstealing in mind. As described in II-B, the `construct_event_set` function might need to scan the entire event queue of the stolen core to build the set of events to be stolen. On our test platform (see Section V for details), the time required to scan a single event in the list (i.e. to follow a link in the list and to check the color of the next

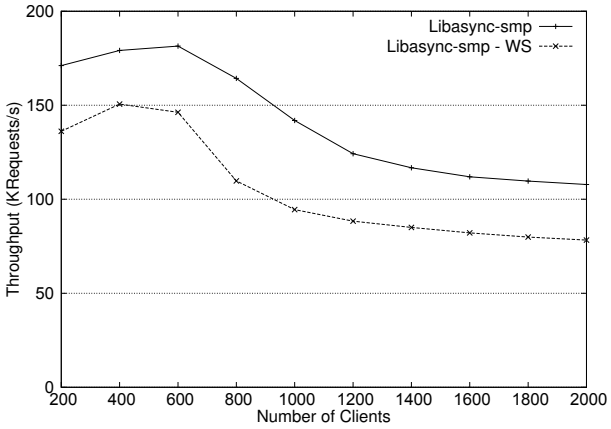

Figure 4. Performance of the SWS Web server with and without workstealing algorithm.

event) is about 190 cycles. This explains why the stealing cost can become very significant when the number of events stored in queues is high. For instance, in the case of the Web server, the most highly loaded cores had on average more than 1000 pending events. These results show that it is crucial to reduce the stealing costs.

III. Improved workstealing algorithm

In this section, we present three complementary heuristics aimed at improving the efficiency of the workstealing algorithm, by making good decisions in the `construct_core_set`, `can_be_stolen` and `choose_color_to_steal` functions introduced in Section II-B. These heuristics have two main goals. First, they aim at improving cache usage by leveraging cache locality between cores on a same die (*locality-aware stealing*), and taking into consideration the size of the data sets accessed by events (*penalty-aware stealing*). Second, they aim at ensuring that it takes less time to steal a set of events than to execute it (*time-left stealing*).

A. Locality-aware stealing

The heuristic presented in this section aims at improving the quality of the victim choice implemented in the `construct_core_set` function. This heuristic is based on the observation that the hierarchy of caches has a huge impact on the performance of multicore processors. Some of these caches are dedicated to one core, some others are shared by a subset of the cores. For instance, in 4-core Intel Xeon processors, cores are divided in 2 groups of 2 cores. Each core has a private L1 cache and shares a L2 cache with the other core in its group. The AMD 16-core architecture features 4 groups of 4 cores. Each core has private L1 and L2 caches, and shares a L3 cache with the 3 other cores in its group. In addition, memory accesses between groups are not uniform [13].

It is thus becoming crucial to design algorithms that take the memory hierarchy into account. Stealing costs highly depend on the *distance* between the stealing and the victim cores.

Table II
Memory access times on an Intel Xeon E5410 machine

Memory hierarchy level	Access time (cycles)
L1 cache	4
L2 cache	15
Main memory	110

Table II shows the latency of the various levels in the memory hierarchy of the machine described in Section V-A. We notice that accessing the event queue of a distant core can be up to 7.3 times slower than for a neighbor core (ie. a core sharing a L2 cache). A similar observation can be made on the time required to access the data set associated with an event (i.e. the data items encapsulated in or referenced by a continuation) stored on a distant queue.

The locality-aware stealing heuristic aims at improving cache usage by minimizing the costs of cache misses. To this end, the `construct_core_set` function returns a set of cores ordered by their distance from the stealing core. [3].

B. Time-left stealing

As we highlighted in Section II, migrating an event from one core to another is costly. This is notably because that stealing requires locking the victim core queue. The time-left heuristic aims at making more relevant decisions on whether cores should be chosen as victims or not. For this purpose, the processing time of events is taken into account.

More precisely, the time-left heuristic consists in dynamically classifying colors into two sets: a set of colors that are worth stealing and a set of colors that should not be stolen. We define a *worthy color* as a color such that the processing time of the set of events associated to that color is superior to the time it would take to steal the set. The function `can_be_stolen` is modified to return true only if such a color exists for a given core. This heuristic requires knowing the average time it takes to steal one single event. This can be known by profiling the runtime. The time-left heuristic also requires knowing the average processing time of the various handlers. This can be achieved by first profiling the application and then annotating the code of handlers.

C. Penalty-aware stealing

This heuristic aims at improving the choice of the color to be stolen. The time-left heuristic described in the previous section relies on the temporal properties of event handlers to classify colors as *worthy* or not. The penalty-aware heuristic aims at choosing the best color from a set of *worthy* colors based on the memory usage of events associated with each color.

The underlying idea can be explained as follows. Events whose handlers access a small data set are good candidates for being stolen since their execution will not introduce substantial cache misses and cache pollution on the stealing core. In contrast, the case of event handlers accessing large data sets

[3]This knowledge can be obtained from the operating system and/or measurements performed at the start of the runtime.

requires a more detailed inspection. If the data set is short lived (e.g. when a handler allocates a buffer and frees it before its completion), then stealing the corresponding events can improve parallelism and does not increase the overall number of cache misses. However, event associated with large data sets that are long-lived (e.g. passed, by value or reference, from a handler to another one) are not good candidates for being stolen. Indeed, migrating such events on distant cores might cause high cache miss rates.

The penalty-aware heuristic allows the application developer to set stealing penalties on event handlers. Events processed by handlers with a high stealing penalty will less likely be stolen than events with a low stealing penalty. This penalty mechanism allows artificially reducing the "attractiveness" of events accessing large, long-lived data sets. In the current state of our work, these annotations are set by the developer based on feedback from application profiling. An underlying assumption is that a given event handler has a relatively stable execution time. This hypothesis is reasonable in our context for two complementary reasons: (i) the small granularity of the considered tasks, and (ii) the effects of the locality and penalty aware strategies, which limit fluctuations caused by cache misses.

IV. The Mely runtime

In this section, we present Mely, an event-based multicore runtime that relies on the event-coloring paradigm. Mely has been designed so as to minimize event stealing costs and implements the three heuristics presented in the previous section. While Mely is backward compatible with Libasync-smp, it differs from it in the workstealing algorithm and in the implementation strategies for storing and managing events. We start with a description of the design of the Mely runtime. Then we discuss the implementation of the workstealing algorithm. Finally, we provide some additional implementation details.

A. Design

Similarly to Libasync-smp, each core runs a single thread in charge of executing event handlers. However, Mely rethinks the way events are manipulated by cores. To drastically reduce the processing time of various workstealing functions like `construct_event_set`, Mely groups events with the same color in distinct queues, called `color-queue`.

Each core maintain a list of `color-queues` which are chained together using a doubly-linked list, called a `core-queue`. Figure 5 depicts the architecture of the Mely runtime that is running on each core (the notion of `stealing-queue` is described in Section IV-B).

Using this organization, a core chooses the next event to be processed by simply taking the first event stored in the first `color-queue`. To prevent starvation, a core is not allowed to indefinitely process events with the same color. There is thus a threshold that defines the maximum number of events with the same color that can be batched processed[4]. In all experiments presented in this paper, the threshold is set to 10. When a `color-queue` is empty, it is removed from the `core-queue`.

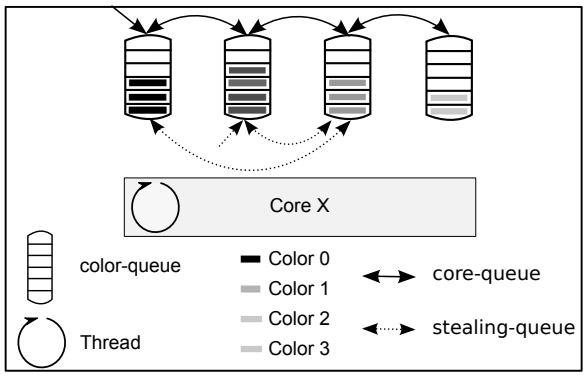

Figure 5. Mely runtime architecture.

When registering a new event, the producing core must first retrieve the adequate `color-queue`. To that end, like Libasync-smp, Mely uses a small (64KB), statically allocated array that keeps track of mappings between colors and `core-queues`. Moreover, if not already present, the producing core also inserts the `color-queue` into the `core-queue` of the core it belongs to.

Accesses to `color-queues` and `core-queues` must be done in mutual exclusion. To that end, as in Libasync-smp, each core owns a spinlock that is used by the different cores when accessing their `color-queues` and `core-queues`. Note that we cannot use a spinlock per color. Indeed, that would not guarantee mutual exclusion when accessing the `core-queues`. Moreover, it is important to outline that a runtime relying on *event-coloring* for managing multiprocessor concurrency cannot store events using DEqueue structures [14] (as often advised in other workstealing-enabled runtimes). The reason is that these structures make the assumption that only one thread registers events in a given queue. In the *event-coloring* approach, several cores can simultaneously try to register events in the queue of any given core.

B. Workstealing implementation

Mely's workstealing implementation is based on Libasync-smp which has been extended to add the locality-aware, time-left, and penalty-aware heuristics. In this section, we detail the implementation of these heuristics.

a) Locality-aware stealing.: The implementation of this heuristic is straightforward ; the `construct_core_set` function build the core set with respect to the cache hierarchy. We use the reification of the cache hierarchy provided by the Linux kernel and made accessible in the `/sys` file system. More precisely, Mely builds a cache map at startup time, that allows each core to discover its neighbors.

b) Time-left stealing.: The implementation of this strategy relies on the use of one `stealing-queue` per core (see Figure 5). These lists store the set of `color-queues` representing *worthy* colors. Within a `stealing-queue`,

[4]When the threshold is reached, the runtime carries on with the next `color-queue` in the `core-queue`.

`color-queues` are ordered according to the cumulative processing time of all events they store. Note that, in order to reduce insertions costs, the `stealing-queue` is only partially ordered: the queue is split in three time-left intervals. Within an interval, `color-queues` are not ordered. This allows balancing insertion and lookup costs in a `stealing-queue`.

When a new event is inserted in a `color-queue`, the cumulative processing time of the queue is incremented accordingly. Symmetrically, when an event is removed from a `color-queue`, its cumulative processing time is decremented accordingly. When a color becomes *worthy*, the corresponding `color-queue` is inserted in the `stealing-queue`. The opposite operation is executed when a color is no longer *worthy*.

As explained in Section III, in the current state of our work, the average processing time of each event handler is provided by the programmer after a profiling phase. The time required to steal an event is obtained from the runtime built-in monitoring facilities.

c) Penalty-aware stealing.: The implementation of the penalty-aware heuristic required defining an annotation allowing the user to set the *workstealing penalty* of each event handler. This penalty is used when computing the cumulative processing time of each `color-queue`. When an event is inserted in a `color-queue`, rather than increasing the cumulative processing time by the processing time of the event, it is increased by the following value: $\frac{event\_time}{ws\_penalty}$. Consequently, an event with a high workstealing penalty will be perceived as requiring less processing time than it actually does.

C. Additional implementation details

Mely is currently based on Gcc 4.3 and Glibc 2.7. Threads are pinned on cores using the `pthread_setaffinity_np` function. We have carefully optimized placement using padding (ie. dedicating one or more cache lines) of private data structures to prevent false sharing. TCMalloc [15] is also used for efficient and scalable memory allocation, reducing contention and increasing spatial locality with per-core memory pools. Lastly, to improve its scalability and robustness, Mely's main event loop for managing network and file I/O replaces the `select()`-based implementation of Libasync-smp with the `epoll` Linux system call, while preserving a compatible API with legacy applications developped for Libasync-smp. [5]

Note that, to provide a fair comparison in the evaluation performed in Section V, we also backported these optimizations inside the legacy Libasync-smp runtime.

V. Evaluation

In this section, we evaluate the Mely runtime. We first describe our experimental testbed. Then, we present microbenchmarks to analyze the individual effects of the heuristics presented in Section III. Finally, we study the performance of Mely using two real-sized system services: a Web server and the SFS file system.

[5]The performance gain brought by the *epoll* system call has been previously observed in the context of highly loaded servers [16].

A. Experimental settings

The experiments are performed on a 8-core machine with two quad-core Intel Xeon E5410 *Harpertown* processors. Each processor is composed of 4 cores running at 2.33GHz and grouped in pairs. A pair of cores from a same processor share a 6 MB L2 cache. Consequently, each processor contains 12 MB of L2 caches. Memory access times are uniform for all cores. The machine is also equipped with 8 GB of memory and eight 1Gb/s Ethernet network interfaces.

For the server experiments, we use between 8 and 16 dual core Intel T2300 machines acting as load injection clients. All machines are interconnected using a Gigabit Ethernet non-blocking switch.

All machines run a Linux 2.6.24 kernel, with hardware counter monitoring support. Runtime and applications are compiled using GCC 4.3.2 with the -O2 optimization flag and run under Glibc 2.7. For all benchmarks, we observe standard deviations below 1%.

B. Microbenchmarks

We use a set of microbenchmarks to study the performance of Mely. We first evaluate the impact of the runtime design on the behavior of the base workstealing (i.e. the workstealing algorithm defined in Libasync-smp). Then, we study the impact of the three workstealing heuristics.

d) Base workstealing.: To evaluate the benefits provided by the careful data placement and the new queue structure, we compare Mely's performance to that achieved by Libasync-smp when enabling and disabling the base workstealing. We use a microbenchmark, called *unbalanced* that works as follows. It implements a fork/join pattern: at each round, 50000 events are registered on the first core. 98% of these events are very short (100 cycles), whereas the other events are much longer (between 10 and 50 Kcycles). Events are independent (i.e. they are registered with different colors and can thus be processed concurrently). When all events have been processed, a new round begins. We repeat this operation during 5 seconds and measure the number of events processed per second.

Table III
IMPACT OF THE BASE WORKSTEALING.

Configuration	KEvents/s	Locking time	WS cost (cycles)
Libasync-smp	1310	0.93%	-
Libasync-smp - WS	122	39.73%	28329
Mely	1265	0.89%	-
Mely - base WS	1195	1.42%	2261

Results are presented in Table III. The *unbalanced* microbenchmark highlights the very bad results of the Libasync-smp workstealing implementation when the input load is not balanced. In particular, we notice that a core, on average, locks its victim for 28 Kcycles, and only steals a set of events requiring 484 cycles to be processed. Moreover, we observe that almost 40% of the time is spent in runtime

locks. As a consequence, the base workstealing algorithm strongly hurts the performance of Libasync-smp (-90%). This microbenchmark also shows that Mely drastically mitigates the performance hit of the base workstealing algorithm. More precisely, it allows reducing the stealing time by a factor of 12.5. However, we can notice that the base workstealing also decreases performances (-5.5%). This highlights the need for smarter stealing heuristics.

e) Time-left stealing.: We evaluate the time-left heuristic using the previously described *unbalanced* microbenchmark. We measure the number of events processed per second when using different workstealing algorithms. Results are presented in Table IV. The time-aware workstealing allows an improvement of 70% over the base workstealing algorithm when executing in Mely. This can be explained by the fact that the time-left heuristic refrains from stealing color sets with a low or negative yield.

Table IV
IMPACT OF THE TIME-LEFT HEURISTIC.

Configuration	KEvents/s	Stolen time (cycles)
Libasync-smp	1310	-
Libasync-smp - WS	122	484
Mely - base WS	1195	445
Mely - time-aware WS	2042	49987

f) Penalty-aware Stealing.: We evaluate the penalty-aware heuristic using a microbenchmark called *penalty*. This microbenchmark works as follows: a single core starts with many events of type A (i.e. events which trigger handler A) associated to different colors, while the other cores start with an empty event queue. When an event of type A is processed, an event of type B with the same color is created. Moreover, the event of type A creates an array fitting in the core cache. Each event of type B accesses an offset of its parent array and registers a new event of type B with the same color. This operation is repeated until the array has been completely accessed. This way, each core executes a set of events with the same color that access the same array. In this benchmark, idle cores have more opportunities to steal events of type B but should preferably steal events of type A to preserve cache locality.

We measure the total number of tasks treated by second. Results are presented in Table V. The penalty of events of type B was set to 1000. We first observe that Libasync-smp achieves very low performance when workstealing is enabled. In constrast, the penalty-aware workstealing allows improving performance by 53% with respect to the Mely runtime executing the base workstealing. These results can be explained by the following fact: the load is initially unbalanced (all events of type A are registered on the same core) and the penalty-aware workstealing allows balancing the load, while keeping a low number of L2 cache misses. Indeed, the number of L2 cache misses per processed event is 95% lower than when executing the base workstealing algorithm in Mely.

g) Locality-aware stealing.: We evaluate the locality-aware heuristic using a microbenchmark called *cache efficient*.

Table V
IMPACT OF THE PENALTY-AWARE STEALING.

Configuration	KEvents/s	L2 misses/Event
Libasync-smp	1103	29
Libasync-smp - WS	190	167K
Mely - base WS	1386	42K
Mely - penalty-aware WS	2122	2K

This microbenchmark uses a fork/join pattern. At each round, one core per pair of cores starts with a hundred events of type A. The handlers for these events allocate an array fitting in their cache and register two events of type B, associated to different colors, on the same core. These events will sort the first and the last part of the array (this mimics the beginning of a merge sort). Once the handler of an event of type B has finished sorting its array, it registers a synchronization event of type C. When the two events of type C registered on each array have been processed, the final part of the merge sort occurs.

Results presented in Table VI show that the locality-aware heuristic allows increasing the performance by 31%. This is explained by the fact that this heuristic allows balancing the load on cores on which no event of type A are initially registered, while ensuring that handlers accessing the same array are executed on neighbor cores. This results in a decrease of L2 cache misses per event of about 83% with respect to the version running the base workstealing.

Table VI
IMPACT OF THE LOCALITY-AWARE STEALING.

Configuration	KEvents/s	L2 misses/Event
Libasync-smp	1156	0
Libasync-smp - WS	1497	13
Mely - base WS	1426	12
Mely - locality-aware WS	1869	2

C. System services

In this section, we evaluate our propositions on two real-sized system services. The first one is a Web server, SWS, which mostly runs short duration handlers for processing requests. The second use case is SFS [12]. Unlike the Web server, SFS mainly executes coarse grain handlers (i.e. cryptographic operations). In both cases, we compare the Mely runtime (with workstealing enabled) and Libasync-smp with and without workstealing.

1) SWS Web server: SWS handles static content, supports a subset of HTTP/1.1, builds responses during start-up (an optimization already used in Flash [6]), and handles errors cases.

The architecture of SWS is similar to the one described by Zeldovich et al. in their initial work on Libasync-smp [11]. However, we optimized cache-management since our workloads always fit in main memory.

The architecture of SWS (illustrated in Figure 6) is similar to the one described by Zeldovich et al. in their initial

work on Libasync-smp [11]. However, we optimized cache-management since our workload fits in main memory.

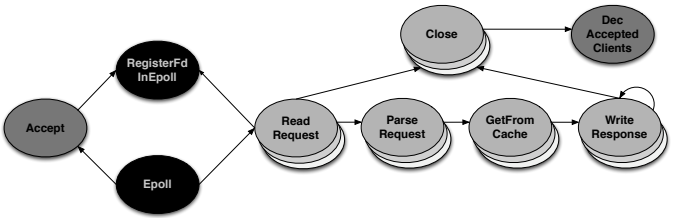

Figure 6. SWS architecture

SWS is structured in 9 event handlers. The *Epoll* is responsible of monitoring active file descriptors. When a file descriptor has pending operations, it registers an event for either the *Accept* or the *ReadRequest* handlers. *Epoll* is always associated with color 0 (thus initially executing on the first core). The *Accept* handler is in charge of accepting new connections. Like in other Web servers, it is possible to specify the maximum number of simultaneous clients. Events associated with this handler are colored with color 1 (thus initially set on the second core). The *ReadRequest* handler is in charge of reading requests. The *RegisterFdInEpoll* handler allows to monitor a new file descriptor. In order to manage concurrency, this handler is colored like *Epoll*. The *ParseRequest* handler is used to analyze the client request. The *CheckInCache* handler gets the response from a map indexed by filename and containing pre-built responses. The *WriteResponse* handler sends responses to the client and the *Close* handler shuts down connections. Finally, the *DecClientAccepted* handler decrements the current number of accepted clients after closing a connection. This handler is colored like *Accept* to manage concurrency.

ReadRequest, ParseRequest, WriteResponse and *Close* events are colored in such a way that requests issued by distinct clients can be concurrently served. For this purpose, we use the file descriptor number of the socket as the color.

For load injection, we developed an event-based closed-loop load injector [17] similar to the one described in [18]. It uses a master/slave scheme, i.e. a master node synchronizes a set of load injection nodes (each simulating multiple HTTP clients) and collects their results.

We evaluate the Mely runtime on SWS when serving small static files of 1KB size. We use 8 physical clients which emulate between 200 and 2000 virtual clients. Each virtual client repeatedly connects to the Web server and requests 150 files. One run lasts 30s and is repeated 3 times.

Figure 7 presents the throughput observed with three runtime configurations: Libasync-smp with workstealing, Libasync-smp without workstealing and Mely with workstealing enabled (with all heuristics activated). To assess the performance of SWS, we also include results for two other efficient and well-established Web servers: the *worker* (multithread) version of Apache [19] and a multiprocess configuration of the event-based μserver [1]. We observe that SWS running on Mely outperforms all the other configurations. μserver shows

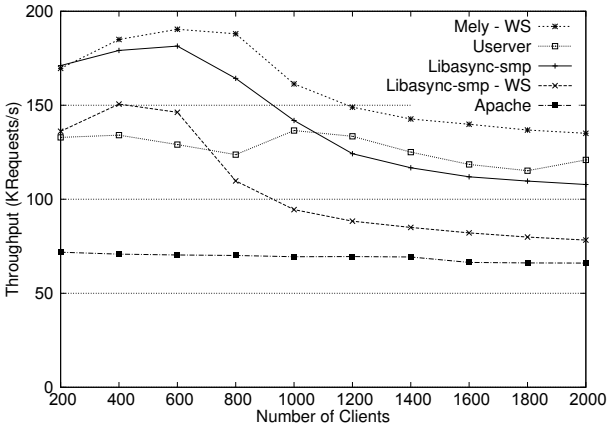

Figure 7. Performance of SWS.

that the *N-Copy* approach has good performances. However, as explained in Section VI, this approach is not always applicable.

In Libasync-smp, enabling the workstealing algorithm decreases performance under this workload by up to 33%. As explained in Section II, this degradation is due to two main factors: (i) very high stealing costs (197 Kcycles) that are superior to the stolen processing time (20 Kcycles), (ii) a drastic increase in L2 cache misses (+146%) over Libasync-smp without workstealing.

Mely outperforms Libasync-smp with workstealing by up to 73%. It steals 14% more processing time (23 Kcycles) and is 32 times faster to steal (6K cycles), thus achieving workstealing efficiency. Moreover, profiling indicates that the locality- and penalty-aware optimizations decrease the number of L2 cache misses by 24%. Mely also improves performance by nearly 25% compared to the Libasync-smp runtime without workstealing. Profiling reveals that the workstealing mechanism relieves the core in charge of the *Epoll* handler from request processing and thus helps improving responsiveness to the incoming network activity.

Additionally, we measure the performance of Mely with workstealing disabled. This configuration has slightly lower performance than Libasync-smp without workstealing (between -7% to -20%) mainly due to the use of many short-lived colors for each connection. Indeed, these short-lived colors introduce costly insertion and removal operations of `color-queues` in `core-queues`. This result emphasizes the efficiency of the Mely workstealing algorithm.

2) Secured File Server (SFS): SFS is an NFS-like secured file system. SFS clients communicate with the server using persistent TCP connections. As all communications are encrypted and authenticated, SFS is CPU-intensive. Our experiments showed that the SFS server spends more than 60% of its time performing cryptographic operations, confirming previous results [11].

We used the coloring scheme described in Libasync-smp [11] where only the CPU-intensive handlers are colored. We performed load injection using 16 client nodes connected to the server through a Gigabit Ethernet switch. Since SFS

only supports a single network interface, we use interface bonding [20] in order to exploit all the available Ethernet ports on the server. Each machine runs a single client that sends requests using the SFS protocol. We use the multio benchmark [21] configured as follows: each client reads a 200MB file. Note that similarly to the benchmark described by Zeldovich et al. [11], the content of the requested file remains in the server's disk buffer cache. Moreover, each client flushes its cache before sending a file request in order to ensure that the request will be sent to the SFS server. Each client computes the throughput at which it reads the file. A master is in charge of collecting the values computed by all the clients.

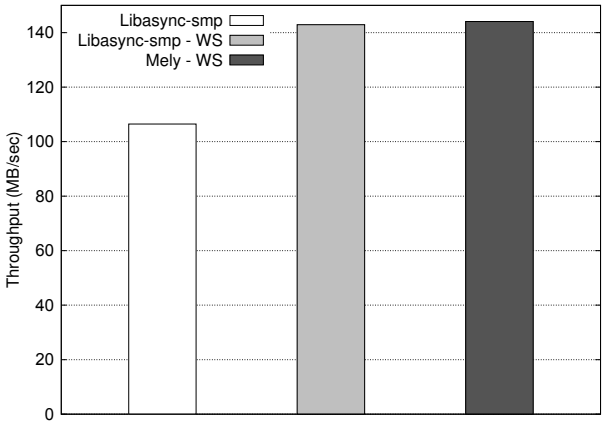

Figure 8. Performance of SFS.

The average throughput is depicted in Figure 8. We evaluate Libasync-smp without workstealing, Libasync-smp with workstealing enabled, and Mely with our improved workstealing algorithm (with all heuristics enabled). As mentioned in Section II, we notice that the legacy Libasync-smp workstealing significantly improves the performance of the SFS server (around 35%). Finally, we observe that Mely's improved workstealing performs similarly to the Libasync-smp workstealing. As expected (see Section II), Mely's workstealing algorithm does not degrade the performance on applications for which the Libasync-smp workstealing is efficient.

VI. Related work

Similarly to the initial publication about Libasync-smp [11], this paper is not aimed at reviving the debate on the relative merits of the thread-based and event-driven models [22], [9], [23], [6], [24], [25], nor on proposing new ways to deal with concurrency and state management issues [22], [26], [10], [27], but focuses instead on improving the performance of existing event-driven software on multicore platforms.

In addition to event-coloring, two other techniques have been used for running event-driven code on parallel hardware. The first one, named *N-copy*, consists in running several independent instances of the same application. While straightforward, such a configuration may reduce efficiency and does not work if the different instances must share mutable state [11]. The second option is based on a hybrid, *stage-based architecture*, combining threads and events: an application is structured as a set of stages interacting via events. Inside a stage, events are executed by a pool of threads [28], [29]. This solution does not suffer from the issues of the N-copy approach but exposes the complexity of preemptive thread-based concurrency to the programmer.

The multiprocessor performance of runtime systems based on structured event queues has been studied, yet with different assumptions regarding the exposed programming model [29] or the application domain and the granularity of tasks [30]. In SEDA, task dispatching decisions are offloaded to the OS thread scheduler and, as far as we know, this aspect has not been studied in details. Due to specific design constraints mentioned by its authors, SMP Click cannot rely on workstealing for adaptive load balancing and uses another custom technique. The applicability of the latter approach to Libasync-smp is limited by the fact that they do not implement the same form of parallelism.

Jannotti et al. [31] have improved and partially automated the specification of mutual exclusion constraints with the event-coloration technique, in order to allow more parallelism. This work is complementary to ours since it is an enhancement of the programming model, for which we present an efficient execution runtime. However, to the best of our knowledge, their proposal has not been fully implemented nor evaluated.

Previous research on uniprocessor event-driven Web servers has demonstrated the benefits of careful event scheduling policies. First, Brecht et al. [32] have shown that tuning the batch scheduling factor of connection-accepting handlers could yield important throughput improvements. Second, Bhatia et al. [33] have highlighted the improved cache behavior provided by interactions between the event scheduler and the memory allocator. We are currently considering how such optimizations can be fruitfully combined with the mechanisms introduced in this paper.

Our context (event-coloring runtimes) brings constraints that are usually not taken into account by the previous studies on workstealing [34], [35] in runtimes like Cilk [36]. These constraints apply to both the runtime data structures and the stolen tasks selection. In particular, we cannot benefit from the use of efficient DEqueues employed in many workstealing-enabled systems [14], [37]. Besides, due to the very small granularity of most tasks in our context, the workstealing costs have a much stronger impact.

McRT [38], the Intel manycore runtime, can also use workstealing for load balancing cooperatively scheduled tasks. However, to the best of our knowledge, it differs from our contribution in several ways. First, it relies on other concurrency control mechanisms such a software transactional memories, which frees the scheduler from the kind of constraint induced by event-coloring. Second, it targets future, very large scale architectures (up to 128 cores, each with multiple hardware threads) using a simulator and thus adopts different tradeoffs (for instance, stealing attempts are restricted to neighbor cores). In contrast, we run our experiments on currently available medium scale hardware. Finally, its evaluation was

focused on desktop rather than server applications.

VII. Conclusion

Event-driven programming is a popular paradigm that has proven well-adapted to the design of networked applications. The event-coloring approach allows such systems to leverage the pervasive hardware parallelism provided by multicore architectures. We study the workstealing mechanism used by Libasync-smp for balancing event processing on cores and show that it can degrade the performance of certain applications such as Web servers.

To overcome these performance issues, we introduce a novel runtime, Mely, which is backward-compatible with Libasync-smp. Mely features an internal architecture aimed at minimizing the cost of workstealing and relies on heuristics to improve the efficiency of stealing decisions. These optimizations can be mostly transparent for application programmers and yield significant performance improvements (up to +73% compared to Libasync-smp with workstealing and +25% compared to Libasync-smp without workstealing). In the worst case, Mely's workstealing does not degrade performance. While our experimental work has focused on the context of Libasync-smp, we believe that our contributions are more general and could be easily applicable to other event-driven runtimes, should they be made multiprocessor-compliant.

As future work, we plan to study techniques to dynamically set time-left annotations and workstealing penalties based on automated monitoring of the running time and memory usage of each handler.

Acknowledgments

This work was partially funded by the OMP European project (FP7-ICT-214009) and the Aravis (Minalogic competitive cluster) project.

References

[1] "The μserver project," 2007, http://userver.uwaterloo.ca.
[2] Acme Labs, "thttpd: Tiny/turbo/throttling http server," http://www.acme.com/software/thttpd/.
[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in *SOSP*, 2001.
[4] M. J. Freedman, E. Freudenthal, and D. Mazières, "Democratizing Content Publication with Coral," in *NSDI*, 2004.
[5] M. Krohn, "Building Secure High-Performance Web Services with OKWS," in *USENIX ATC*, 2004.
[6] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable Web server," in *USENIX ATC*, 1999.
[7] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris, "OverCite: A Distributed, Cooperative CiteSeer," in *NSDI*, May 2006.
[8] Zeus Technology, "Zeus Web Server," http://www.zeus.com/products/zws/.
[9] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-Driven Programming for Robust Software," in *ACM SIGOPS European Workshop*, 2002.
[10] M. Krohn, E. Kohler, and M. F. Kaashoek, "Events Can Make Sense," in *USENIX ATC*, 2007.
[11] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and M. F. Kaashoek, "Multiprocessor Support for Event-Driven Programs," in *USENIX ATC*, 2003.
[12] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating Key Management From File System Security," in *SOSP*, 1999.
[13] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *OSDI*, 2008.
[14] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *SPAA*, 2005.
[15] S. Ghemawat and P. Menage, "TCMalloc : Thread-Caching Malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
[16] D. Kegel, "The c10k problem," 2006, http://www.kegel.com/c10k.html.
[17] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: a Cautionary Tale," in *NSDI*, 2006.
[18] G. Banga and P. Druschel, "Measuring the Capacity of a Web Server," in *USITS*, 1997.
[19] "The Apache HTTP server project," http://httpd.apache.org.
[20] The Linux Foundation, "Bonding multiple devices," http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding.
[21] "The multio benchmark," 2004, http://www.cisl.ucar.edu/css/software/multio/.
[22] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management Without Manual Stack Management," in *USENIX ATC*, 2002.
[23] J. K. Ousterhout, "Why threads are a bad idea (for most purposes)," Presentation given at the USENIX ATC, 1996.
[24] R. von Behren, J. Condit, and E. A. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *HOTOS*, 2003.
[25] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *SOSP*, 2003.
[26] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner, "Flux: A Language for Programming High-Performance Servers," in *USENIX ATC*, 2006.
[27] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and Exploiting Concurrency in Networked Applications with Aspen," in *PPoPP*, 2007.
[28] J. R. Larus and M. Parkes, "Using Cohort Scheduling to Enhance Server Performance," in *USENIX ATC*, 2002.
[29] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned scalable internet services," in *SOSP*, 2001.
[30] B. Chen and R. Morris, "Flexible Control of Parallelism in a Multiprocessor PC Router," in *USENIX ATC*, 2001.
[31] J. Jannotti and K. Pamnany, "Safe at Any Speed: Fast, Safe Parallelism in Servers," in *HotDep*, 2006.
[32] T. Brecht, D. Pariag, and L. Gammo, "Acceptable Strategies for Improving Web Server Performance," in *USENIX ATC*, 2004.
[33] S. Bhatia, C. Consel, and J. L. Lawall, "Memory-Manager/Scheduler Co-Design: Optimizing Event-Driven Servers to Improve Cache Behavior," in *ISMM*, 2006.
[34] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.
[35] F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *FPCA*, 1981.
[36] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.
[37] M. Herlihy and N. Shavit, "Chapter 16: Futures, Scheduling and Work Distribution," in *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008, pp. 369–396.
[38] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling Scalability and Performance in a Large Scale CMP Environment," in *EuroSys*, 2007.