Leveraging Hardware Message Passing for Efficient Thread Synchronization

Darko Petrović Thomas Ropars André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract

As the level of parallelism in manycore processors keeps increasing, providing efficient mechanisms for thread synchronization in concurrent programs is becoming a major concern. On cache-coherent shared-memory processors, synchronization efficiency is ultimately limited by the performance of the underlying cache coherence protocol. This paper studies how hardware support for message passing can improve synchronization performance. Considering the ubiquitous problem of mutual exclusion, we adapt two state-of-the-art solutions used on shared-memory processors, namely the server approach and the combining approach, to leverage the potential of hardware message passing. We propose HYBCOMB, a novel combining algorithm that uses both message passing and shared memory features of emerging hybrid processors. We also introduce MP-SERVER, a straightforward adaptation of the server approach to hardware message passing. Evaluation on Tilera's TILE-Gx processor shows that MP-SERVER can execute contended critical sections with unprecedented throughput, as stalls related to cache coherence are removed from the critical path. HYBCOMB can achieve comparable performance, while avoiding the need to dedicate server cores. Consequently, our queue and stack implementations, based on MP-SERVER and HYBCOMB, largely outperform their most efficient pure-shared-memory counterparts.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords combining; mutual exclusion; concurrent objects; message passing

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02. . . $15.00.
http://dx.doi.org/10.1145/2555243.2555251

1. Introduction

As industry is shifting toward manycore processors, it is increasingly important to put the constantly growing number of cores to good use. For some types of applications, for instance scale-out workloads typically found in data centers, this is not a problem because of their *embarrassingly parallel* nature. There are, however, applications whose parallelization requires significant effort, as they contain data or objects intensively *shared* by multiple threads. To ensure consistency, threads must access such shared parts of the program state in a synchronized fashion. Whether synchronization is implemented using critical sections (CSes) or nonblocking (lock-free) algorithms, it creates sequential bottlenecks that, because of *Amdahl's law*, ultimately limit application speedup. Indeed, recent studies show that optimizing contended CSes can significantly improve the performance of some workloads [17, 27]. Furthermore, fast synchronization on simple concurrent objects, such as queues, is key to the performance of parallelization frameworks [4]. It is therefore of great importance to understand the subtleties of synchronization and to continue making it more efficient.

At the same time, as parallel programming is becoming mainstream, it is desirable to provide *universal constructions* that enable non-experts to easily write highly-efficient concurrent code. In this work, we study universal constructions for executing contended CSes. The state of the art in this field is the *combining* synchronization technique [10, 11, 13, 24]. The key idea behind combining is that a thread, holding the lock on an object, should not immediately release it after executing its own CS. Instead, the thread executes a number of pending CSes of other threads as well, which minimizes the cost of lock handover and improves data locality. A more extreme version of this idea, sometimes referred to as *delegation* [8], is to earmark a special *server* thread and pin it to a certain processor core. The server thread does not run application code, but only executes CSes of other threads. Dedicating cores is less feasible if an application includes a large number of potentially contended concurrent objects.

A vast majority of work on thread synchronization, including existing universal constructions, assumes the shared memory programming model. Shared memory is often built using local caches and a complex *cache coherence protocol* [26], which makes the caches functionally invisible to the programmer. However, the performance impact of the cache coherence protocol on concurrent algorithms cannot be ignored [11, 13]. Optimizing concurrent code requires in-depth understanding of cache coherence protocols and memory consistency models. Vendors tend to hide their details, or provide them in informal or incomplete ways [23], which makes the task of designing efficient concurrent algorithms notoriously hard. On top of that, the future of cache coherence is uncertain: Some recent studies question the scalability of the traditional cache-coherent shared-memory approach and advocate the use of message passing [6, 16, 29]. As a result, there are experimental [16] as well as commercial [1, 2] processors with hardware support for sending application-level messages between processor cores.

Message passing offers explicit control over communication, so some studies call for complete redesign of software with message passing in mind, notably in the context of operating systems [6, 29]. The same can be advocated for thread synchronization: Indeed, some recent work presents concurrent objects that rely on message passing [8, 20]. A question that arises, however, is whether a full paradigm shift is necessary and justified. Although the problems of cache coherence are evident, we ought to precisely quantify advantages that message passing could provide. Also, even if message passing is advantageous, this does not mean coherent shared memory should be abandoned altogether. In support of this, there are arguments that on-chip cache coherence can scale to large core counts as its overhead in terms of traffic, storage, latency and energy can be made to increase very slowly with the number of cores [18].

Coherent shared memory and message passing coexist in some recent *hybrid* processors, such as Tilera's TILE-Gx processor family [2]. As such, it provides a large design space for synchronization primitives. It is also an ideal testbed to experimentally compare shared memory and message passing approaches. In this work, we consider the problem of contended CSes, and use TILE-Gx to study how hardware message passing can make their execution more efficient than with classic shared-memory techniques.

Our findings indicate that state-of-the-art solutions for efficient CS execution based on a server (RCL [17]) or a combiner (CC-SYNCH [11]) waste much time in CPU stalls resulting from activities related to cache coherence. When CSes are short, these stalls dominate all other overheads. To overcome this problem, we take advantage of hardware message passing and present two solutions: MP-SERVER, a simple server-based approach, and HYBCOMB, a universal construction based on the combining technique. Whereas adapting the server-based approach used in shared-memory systems to message passing is straightforward, the design of HYBCOMB involves significant algorithmic complexity. As its name suggests, HYBCOMB is a *hybrid* algorithm that relies both on cache-coherent shared memory and hardware message passing for synchronization: Hardware message passing is used to exchange requests and responses between the combiner and other threads, while shared memory is used to manage combiner identity (which would be complex and inefficient to do using message passing).

We evaluate the performance of MP-SERVER and HYBCOMB, by implementing ubiquitous linearizable [15] concurrent objects, namely counters, queues, and stacks. Experiments with counters show that MP-SERVER outperforms CC-SYNCH and RCL by up to 4.3x. This is due to the fact that, in high concurrency levels, virtually no stalls remain on the critical path of the server. HYBCOMB also largely outperforms the pure-shared-memory solutions, and can achieve performance close to the one of MP-SERVER, while avoiding the need to dedicate cores. Compared to other queues and stacks, our new implementations on top of MP-SERVER and HYBCOMB reach up to 2x and 1.5x higher throughput respectively, shedding light on the advantages of hardware message passing for synchronization.

In summary, the contributions of this work are the following. We analyze the performance limitations of state-of-the-art solutions for efficient CS execution over cache-coherent shared memory in Section 3. In Section 4, we describe MP-SERVER and HYBCOMB, our two synchronization solutions based on hardware message passing. This includes the full specification and a proof sketch of HYBCOMB, which is, to the best of our knowledge, the first combining algorithm that exploits the hybrid nature of contemporary processors. Finally, we present an extensive evaluation of MP-SERVER and HYBCOMB in Section 5. On the example of linearizable counters, queues and stacks, we show that they perform significantly better than their most efficient known shared-memory counterparts.

2. System Model

We assume a set of T sequential threads that can communicate both by issuing operations to coherent shared memory and by directly exchanging messages.

Cache coherence. In the cache-coherent (CC) shared-memory model, threads operate on cached copies of shared variables. We assume a model adapted from the one by Sorin et al [26]. A processor chip is composed of single-threaded cores. Each core has its local, private data cache. All cores have access to a globally shared memory through an interconnection network. The cache coherence protocol maintains the *single-writer-multiple-reader* invariant: At any given time, either a single core has read-write access to a cached variable, or some cores have read-only access [26]. *Remote Memory References* (RMRs) are accesses to shared variables that involve communication on the interconnection network. In this model and assuming write-back caches, reading a shared variable generates an RMR if the core does not hold a copy of the variable in either mode. Writing a shared variable generates an RMR if the core does not hold a copy of the variable in read-write mode.

Shared-memory operations. The memory is an array of 64-bit locations. Similarly to most related studies, we assume that the memory is sequentially consistent. Supported operations on a memory location a are the standard *read*(a), *write*(a, v) operations as well as some atomic read-modify-write operations, namely *FAA*(a, v) (fetch-and-add), *SWAP*(a, v) and *CAS*(a, v_{old}, v_{new}) (compare-and-set[1]), with their standard definitions.

Message-passing operations. Each thread has its incoming FIFO message queue that stores 64-bit values (message queue hereafter). Supported operations are $send$, $receive$ and is_queue_empty. The operation $send(i, M)$ puts message M, which is a set of values $v_1, v_2, ..., v_n$, in the message queue of thread t_i. The $send$ operation is asynchronous, *i.e.*, it may return before M is placed in the destination message queue. Message transmission time is bounded but unknown, *i.e.*, the time between a call to $send$ and the moment when the message is placed in the corresponding queue is arbitrarily, but finitely long. If $|M| > 1$, values are placed in the destination message queue in the order $v_1, v_2, ..., v_n$. The operation $receive(k)$ returns k values from the head of the local message queue. If there are fewer than k values in the queue, the operation blocks until k values are available. Operation $is\_queue\_empty()$ returns true if the local message queue is empty.

3. Critical Sections over CC Shared Memory

This section details existing techniques for the efficient execution of highly-contended critical sections on cache-coherent processors. It explains how their performance is influenced by the underlying CC protocol.

On a CC processor, the number of RMRs generated by a synchronization algorithm should be minimized. Indeed, an RMR is typically orders of magnitude more expensive than an access to the local cache. This section shows that even the most efficient shared-memory synchronization techniques to implement mutual exclusion on a CS generate a constant number of RMRs per CS execution. Thus, their performance depends on the CC protocol.

Critical sections are usually implemented using locks. In this context, the basic technique to limit the number of RMRs is to introduce *local spinning* [19]: Each thread polls on a different variable which stays in its local cache, to limit the number of RMRs and avoid contention on the interconnection network. Queue locks provide local spinning [5, 19] and achieve an $O(1)$ RMR complexity per lock acquisition. In addition to local spinning, locality inside the CS can be optimized to further reduce the number of RMRs. The key idea is that, instead of moving the data associated with a CS to the core that wants to execute the CS, the CS is executed on the core where the data are located. If contention is high, this results in a substantial performance increase over classic locks. We can identify two approaches that exploit this idea: the client-server approach [9, 17], and the combiner approach [10, 11, 13, 24].

[1] The variant of compare-and-swap that returns a boolean.

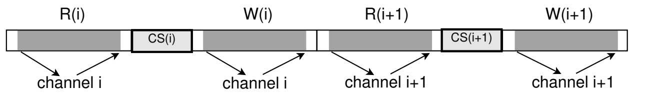


Figure 1. Mutual exclusion server – shared-memory implementation; $R(i)$, $W(i)$ – resp. reading from, writing to the channel of client i; $CS(i)$ – corresponding critical section; dark grey – server stalls (due to RMRs)

Remote Core Locking (RCL) [17] is an efficient implementation of the client-server approach. A non-application thread (the server) is in charge of executing CSes. Application threads (clients) send requests to the server to execute a critical section on their behalf. Assuming that data accessed inside the CSes are never accessed by application threads outside the CSes, these data remain in the cache of the server, ensuring that the number of RMRs during CS execution is minimized. Ideally, the only RMRs that remain on the critical path of the CS execution are the ones related to synchronization between the clients and the server. Figure 1 illustrates the execution of an RCL server. For client-server communication in RCL, each client thread has a dedicated cache line, which it uses as a bi-directional channel. When client i wants to execute a CS, it writes its request to the cache line $channel_i$, and then spins on that cache line until it receives a reply from the server. The server first reads the request from $channel_i$ ($R(i)$ in Figure 1). Since the last access to $channel_i$ was from client i writing the request, this read triggers an RMR (server stalls are represented in dark grey). Then, the server executes the critical section ($CS(i)$). Finally, it writes to $channel_i$ to inform the client that the request has been processed ($W(i)$). This write triggers another RMR to invalidate the client's copy of the cache line. The figure assumes high load, *i.e.*, the server is never idle, and shows that in this case there are two RMRs at the RCL server per CS. Note that Figure 1 is somewhat simplified, since it assumes sequential consistency. On a real processor, the different RMRs might partially overlap, depending on the memory consistency model of the processor at hand, resulting in fewer CPU stalls. Nevertheless, these RMRs remain an important source of overhead even on a processor with weak memory consistency (see Section 5).

While keeping similar performance benefits, the combiner approach does not require dedicated servers [13]. When a thread gets a lock associated with a CS, it becomes a *combiner* and executes operations of other threads that are waiting to access this CS, in addition to its own. To prevent the combiner from starving if the number of operations of other threads to execute is high, the combiner role is handed over to another thread when the current combiner has served a predefined number of requests. CC-SYNCH [11] is, to our knowledge, the most efficient combiner-based approach. Since the combiner changes over time, the synchronization mechanism is more complex than in RCL. Nevertheless, while a thread is acting as a combiner, CC-SYNCH is similar to RCL with respect to RMRs: It generates one RMR to read a request from another thread, and then generates another RMR to inform that thread that the operation has been performed.

The server-based approach has the advantage of being simple and very efficient in cases where a small number of clearly identified CSes are highly contended [17]. On the other hand, combining is more flexible, which comes at the expense of requiring more complex synchronization between threads. Indeed, combiners adapt themselves automatically to the load: If a CS is highly contended, all the CPU cycles of one core will be temporarily allocated to it, but if no thread tries to execute a CS, no resources are consumed.

Both with RCL and CC-SYNCH, only two RMRs related to thread synchronization remain on the critical path of a CS execution. These two RMRs, however, can have a big impact on throughput if the code to execute in the CS itself contains few or no RMRs.

4. Critical Sections using Message Passing

We present two ways to leverage hardware support for message passing to execute critical sections efficiently. Taking the server approach, we first explain why hardware messaging can be beneficial in this context. Then we present a novel combining algorithm that uses both shared memory and hardware message passing for thread synchronization.

4.1 The Server Approach (MP-SERVER)

A client-server approach, such as RCL, is a natural fit for message passing. Indeed, RCL's client-server communication layer can be seen as an implementation of message passing over shared memory. Instead, we simply leverage hardware message passing support to implement client-server communication. We refer to this solution as MP-SERVER. Based on the model introduced in Section 2, Figure 2 explains why MP-SERVER may have better performance than its shared-memory counterpart. Compared to Figure 1, stalls can be avoided for two reasons. First, the server reads requests from the local message queue, without any remote actions that would cause it to stall. Second, the server does not wait for the actual message transmission to take place when it sends a response. When and how the messages are actually sent to their destinations is the responsibility of the underlying hardware message passing implementation. Therefore, if hardware message passing is used, we expect to be able to *completely remove* stalls related to synchronization from the critical execution path.

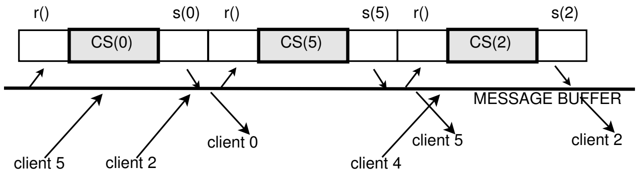


Figure 2. Mutual exclusion server – message-passing implementation; $r()$ – receive message; $s(t)$ – send message to thread t; request from client 0 is already available in the server's message queue

4.2 The Combiner Approach (HYBCOMB)

We now detail HYBCOMB, our combining algorithm tailored to take advantage of message passing. We start by describing the main principles of combining techniques over shared memory, to identify how message passing can be used to improve performance.

Main principles. In combining algorithms, threads interact for two purposes: (i) *electing* a combiner; (ii) exchanging information between the combiner and threads that have operations to be executed in mutual exclusion. In shared-memory combining algorithms [11, 13, 24], these two tasks are handled by a single shared object: a list of requests. To execute an operation, a thread adds a request to the list. The current combiner traverses the list to fetch and execute requests. When the current combiner wants to return, it hands over the combining role to the thread owning the next request in the list (if there are no requests to be executed, the next thread that inserts a request will become the combiner).

HYBCOMB uses hardware message passing for synchronization between the combiner and the other threads. As long as the combiner does not change, synchronization works as with MP-SERVER (Figure 2). Still, we use shared memory for managing combiner identity. In a nutshell, HYBCOMB works as follows: When a thread t wants to execute a request, it first checks the identity of the combiner through a shared variable. If a combiner is available and ready to handle the request, t sends a message to that combiner. If not, t tries to promote itself to a combiner, by executing CAS on the variable that keeps the combiner identity.

Managing combiner identity using message passing would be complex and probably inefficient. The main problem is that a thread acting as a combiner has to stop combining at some point, which must be synchronized with actions of other threads. To get its operation executed by a combiner, a thread has to get the identity of the combiner thread and send a request to it. If the combiner identity changes in the meantime, the operation will never get executed. Dealing with this problem using message-passing would require either a delegated thread (which is exactly what the combiner approach is trying to avoid), or intensive communication between threads (*e.g.*, broadcast).

Detailed description. Algorithm 1 describes HYBCOMB. The interface is the same as that of CC-SYNCH: When a thread wants to execute a critical section, it calls the $apply\_op$ method, providing a pointer to the function to execute and its arguments.[2] Note, however, that HYBCOMB is not just a simple adaptation of existing combining algorithms, where message passing is used instead of a shared list to make the combiner thread aware of the requests to execute. As already mentioned, using message passing requires us to be able to identify the combiner thread to which requests should be sent. This should be carefully handled, especially at the time the combiner changes. This problem does not exist in combining techniques fully based on shared memory since it is the combiner thread that fetches requests from a shared data structure.

The code executed by the active combiner are lines 23-43. Algorithm 1 ensures that these lines are executed in mutual exclusion, *i.e.*, that there is a single active combiner at a time. To manage combiner identity, a data structure called $Node$ is used. Each thread owns a reference to a different node (my_node). The id of the thread owning a node is saved in the field $Node.thread\_id$. Managing combiner identity is done using the shared pointer $last\_registered\_combiner$. To become a combiner (lines 17-21), a thread t tries to execute a CAS operation on $last\_registered\_combiner$ to make it point to its node. If the CAS succeeds, t keeps a pointer to the node corresponding to the previous $last\_registered\_combiner$ in its local variable $last\_reg$. This mechanism can be seen as building a logical queue where the head of the queue is the current active combiner and the tail is the $last\_registered\_combiner$, each thread in the queue having a reference to the predecessor in its $last\_reg$ variable. The $Node.combining\_done$ flag is used to synchronize the threads in the queue. Before starting executing as a combiner, a thread spins on the $combining\_done$ flag of its predecessor (line 19), which is set by the predecessor when it finishes combining (line 42).

Upon calling $apply\_op$, a thread t first tries to register its request with $last\_registered\_combiner$. It does so by performing a fetch-and-increment on the $Node.n\_ops$ field of the corresponding node. This field guarantees that one combiner will receive and execute at most MAX_OPS requests of other threads. If the threshold MAX_OPS is not reached, t sends its request to the combiner using message passing (line 13), and waits for a response (line 14). If the last registered combiner cannot accept any new request, t tries to register itself as a combiner as already explained.

Once t becomes the active combiner, it first executes its own request (line 23). Then it reads messages from its message queue, processes requests and sends responses. When its message queue is empty, t decides to stop combining and announces it by writing MAX_OPS to its n_ops field.

[2] To make the presentation more concise, the shared-memory object on which the critical section is executed is implicit.

Algorithm 1 HYBCOMB combining algorithm – code for thread id

```
 1: const $MAX\_OPS$ {* max. operations per combiner *}
 2: type $Node\{thread\_id$: int, $n\_ops$: int, $combining\_done$: bool$\}$

Global Variables:
 3: $departed\_combiner: Node \leftarrow \{\perp, MAX\_OPS,$ true$\}$
 4: $last\_registered\_combiner: Node \leftarrow departed\_combiner$

Local Variables:
 5: $my\_node: Node \leftarrow \{id, MAX\_OPS,$ false$\}$

 6: apply_op ($func\_ptr, args$)
 7:     $ops\_completed \leftarrow 0$
 8:     loop
 9:         $last\_reg \leftarrow last\_registered\_combiner$
10:         {* try to register with last registered combiner *}
11:         if $FAA(last\_reg.n\_ops, 1) < MAX\_OPS$ then
12:             {* success. send message to combiner and wait *}
13:             $send(last\_reg.thread\_id, \{id, func\_ptr, args\})$
14:             return $receive(1)$
15:         else
16:             {* failure. try to register as combiner *}
17:             if $CAS(last\_registered\_combiner, last\_reg, my\_node)$
                then
18:                 $my\_node.n\_ops \leftarrow 0$
19:                 while $\neg last\_reg.combining\_done$ do
20:                     $nop$
21:                 break

22:     {* became combiner. do your own op first *}
23:     $retval \leftarrow func\_ptr(args)$

24:     {* as long as message queue is not empty, handle requests *}
25:     while $\neg is\_queue\_empty()$ do
26:         $\{sender\_id, fptr, fargs\} \leftarrow receive(3)$
27:         $send(sender\_id, fptr(fargs))$
28:         $ops\_completed \leftarrow ops\_completed + 1$

29:     {* close combining for new requests *}
30:     $total\_ops \leftarrow SWAP(my\_node.n\_ops, MAX\_OPS)$
31:     if $total\_ops > MAX\_OPS$ then
32:         $total\_ops \leftarrow MAX\_OPS$

33:     {* serve remaining requests *}
34:     while $ops\_completed < total\_ops$ do
35:         $\{sender\_id, fptr, fargs\} \leftarrow receive(3)$
36:         $send(sender\_id, fptr(fargs))$
37:         $ops\_completed \leftarrow ops\_completed + 1$

38:     {* exchange your node, inform next combiner and return *}
39:     $my\_node \leftarrow SWAP(departed\_combiner, my\_node)$
40:     $my\_node.combining\_done \leftarrow$ false
41:     $my\_node.thread\_id \leftarrow id$
42:     $departed\_combiner.combining\_done \leftarrow$ true
43:     return $retval$
```

Since it does so using SWAP, it retains the old value of n_ops (in $total\_ops$), which is the total number of requests it has to serve as a combiner. It then finishes its combining round by serving the remaining requests, if any (lines 34-37).

Before returning, t must get the node it will use next time it calls $apply\_op$ (we want to avoid allocating a new node for every $apply\_op$ call). Obviously, t cannot use the same node because that requires the $combining\_done$ field to be reset, but t cannot know when the next combiner will have read this field. As a solution, only one additional node is allocated for all n threads, and t gets the node that was used by the previous combiner (pointed by $departed\_combiner$) (lines 39-42)[3]: t knows that the $combining\_done$ field of this node can be reset since t was the thread spinning on this node. Finally, note that t must not reset the n_ops field of its new node at this point because other threads might still have an old reference to this node in their $last\_reg$ variable (lines 9-11): if n_ops were reset, these threads could send requests to t while it is not a combiner. Thus, t will reset n_ops only once it registers as a combiner again (line 18).

Additional comments. Before presenting the proof of correctness, we make a few remarks on the way HYBCOMB works. First, we can note that registering as a combiner (line 17) and resetting the n_ops counter (line 18) are not atomic. This does not affect the correctness of the algorithm. In the very unfortunate case where a thread t' executes the FAA at line 11 while t is between those two lines, t' will simply not manage to register its request with t, and so, will try to become the next combiner. This could merely result in a performance penalty as t would only have its own request to execute as a combiner. Results presented in Section 5 show that this rarely occurs in practice.

Note also that the first *while* loop in the request execution part (lines 25 to 28) is not necessary for correctness: The thread can decide to stop combining as soon as it has executed its own request. Still, this loop is beneficial for performance, as postponing the SWAP at line 30 increases the combining potential.

HYBCOMB uses a CAS operation like some other combining algorithms [13, 24], but unlike CC-SYNCH [11]. It is well known that CAS can impair performance (because it can repeatedly fail, causing contention) as well as fairness (a thread can starve if it executes CAS in a loop and persistently fails). We still choose to use CAS and not SWAP at line 17 for the following reasons: i) if SWAP is used and several threads try to register as combiners, they all succeed but some of them only have their own request to execute as a combiner, whereas with CAS only one thread manages to register as a combiner, and potentially execute all other requests; ii) the CAS is not expected to be a hot spot in HYBCOMB as it is only executed when a thread wants to register as a combiner. Experiments presented in Section 5 confirm the second point. If desired, a middle ground would be to use SWAP only if CAS fails several times.

Proof of correctness. Due to the space constraints, we only sketch the proof. The key idea is to show that Algorithm 1 maintains a logical queue of $Nodes$, denoted by CS_{queue}, (queue for entering the CS corresponding to lines 23 to 43) where each node represents a thread. The head of the queue is the current combiner that executes the CS. Other nodes in the queue, if any, correspond to threads that want to become combiners, *i.e.*, to enter the CS. The operation $insert$ into CS_{queue} corresponds to a successful execution of CAS at line 17. The operation $remove$ from CS_{queue} corresponds to the execution of lines 39 to 43. Algorithm 1 maintains the following invariants related to CS_{queue}: (i) the tail of CS_{queue} is the node pointed to by the global variable $last\_registered\_combiner$ (line 4); (ii) $\forall n_t \in CS_{queue}$, node n_t corresponding to thread t, we have $last\_reg_t = n$, n being the predecessor of n_t in CS_{queue}; (iii) if $last\_reg_t.combining\_done = true$, then t is the head of CS_{queue}. We denote these invariants related to CS_{queue} by I_1. In addition to I_1, we consider the following invariants:

- I_2 :: For every thread t, $my\_node_t.thread\_id = id(t)$.
- I_3 :: There is at most one node n such that $n.combining\_done = true$;

Proposition 1. *I_1, I_2, I_3 are invariants of Algorithm 1.*

The proof is as follows. Let us denote by $I(x, y)$ the fact that I_1 to I_3 hold after x executions of $insert(CS_{queue})$ and y executions of $remove(CS_{queue})$. We prove that for all x, y, we have $I(x, y)$ by a double induction on x and y: first, we prove $I(1, 0)$ and $I(1, 1)$ (base step); second, we prove the induction step: $I(x, y) \Rightarrow I(x + 1, y)$ and $I(x, y) \Rightarrow I(x, y + 1)$ (if $x > y$).

Proposition 1 ensures lines 23 to 43 are executed in mutual exclusion (one combiner at a time). Then, the two lemmas

Lemma 1. *If for node n we have $n.n\_ops < MAX\_OPS$, then n is in CS_{queue}.*

Lemma 2. *If the message queue of thread t contains a request, then the node pointed to by my_node_t is in CS_{queue}.*

allow us to prove the following result:

Proposition 2. *At line 14, thread t cannot receive a request (i.e., t can only receive the response to the request sent at line 13).*

Mutual exclusion established by Proposition 1 together with Proposition 2 show that Algorithm 1 is *safe*. The linearizability of Algorithm 1, with respect to calls to $func\_ptr$, follows directly. For *liveness*, we need to prove additional results:

Lemma 3. *Every combiner t executes its own operation (line 23), and all the operations sent to it (at line 13); then t removes itself from CS_{queue}.*

Finally:

Proposition 3 (liveness)**.** *Algorithm 1 is deadlock-free.*

If thread t wants to execute some operation *op*, then either t eventually gets the response (Lemma 3), or t tries to enter CS_{queue} (line 17). In the latter case, if t succeeds (executes CAS successfully), then it eventually executes *op* and leaves CS_{queue} (Lemma 3).

[3] The use of a SWAP operation at line 39 to exchange the two nodes is only for brevity. An atomic operation is not needed since these lines are executed in mutual exclusion anyway.

5. Evaluation

In this section we implement and thoroughly evaluate the algorithms presented in Sections 3 and 4. We begin by introducing the used hybrid processor and our experimental setup. Then we present experiments that evaluate different implementations of a concurrent counter. We then extend our analysis to more complex concurrent objects, namely queues and stacks. Finally, we discuss the generality of our results and their applicability to other platforms.

5.1 Platform

We use the Tilera TILE-Gx8036, which integrates 36 cores, works at 1.2 GHz and features complete hardware support for both coherent shared memory and message passing [2]. Software-wise, we use GCC 4.4.6 and version 2.6.40.38-MDE-4.1.0.148119 of Tilera's custom Linux kernel. The memory consistency model is relaxed compared to x86, so a careful use of memory fences is necessary to avoid inconsistency. Each core has a dedicated hardware message buffer, capable of storing up to 118 64-bit words. The message buffer of each core is 4-way multiplexed, which means that every per-core buffer can host up to four independent hardware FIFO queues, containing incoming messages. The User Dynamic Network (UDN) allows applications to exchange messages directly through the mesh interconnect, without OS intervention. While exchanging messages, a thread must be pinned to a core and registered to use the UDN (but it can unregister and freely migrate afterwards). When a message is sent from core A to core B, it is stored in the specified hardware queue of core B. The *send* operation is asynchronous and does not block, except in the following case. Since messages are never dropped, if a hardware queue is full, subsequent incoming messages back up into the network and may cause the sender to block. It is the programmer's responsibility to avoid deadlocks that can occur in such situations. When a thread executes *receive* on one of the four local queues, the first message from the queue is returned. If there are no messages, the thread blocks. Messages consist of one or multiple words.

5.2 Methodology and Setup

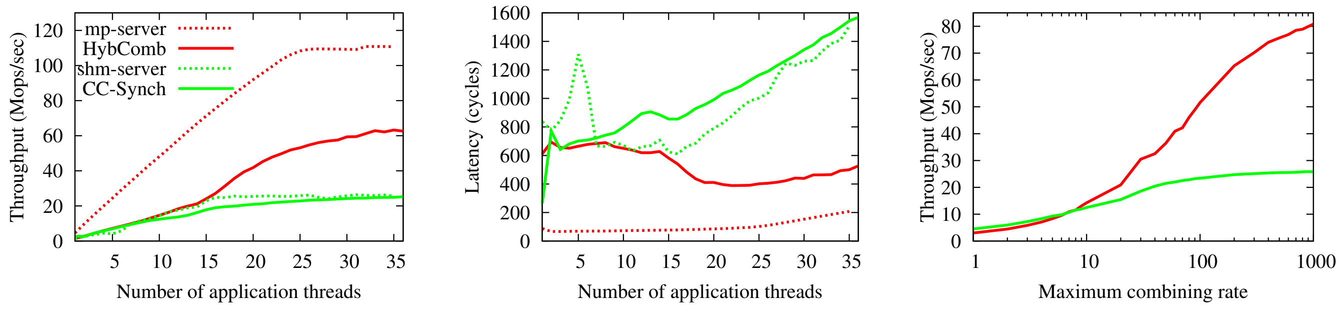
We have implemented MP-SERVER and HYBCOMB on the TILE-Gx, as well as two algorithms purely based on shared memory: the CC-SYNCH combining algorithm [11] and SHM-SERVER, a server approach. SHM-SERVER can be seen as a simplified version of RCL [17], since it implements the same core mechanism (an array of cache lines, one for each client), but lacks support for some advanced features, such as nested critical sections (note that this simplification does not decrease performance). The implementations have been carefully optimized and compiled with the O3 flag. Because of the relaxed memory model of the TILE-Gx, we have inserted memory fences where necessary to ensure correctness. We assume that shared data is accessed only inside CSes, which holds for the concurrent objects we evaluate. A more conservative use of memory fences would be necessary when this is not the case [9]. To obtain the best possible performance, we augment all of the implementations with a simple interface that allows a thread to send a unique opcode of the CS to the servicing thread, rather than a function pointer. This allows the compiler to inline the function calls that the servicing thread makes for every CS, which results in a visible performance increase in most cases [9]. It is worth noting that the results are qualitatively the same without this optimization.

We use the methodology commonly found in related studies [11, 13, 21, 22]: In each experiment, a specified number of application threads repeatedly execute operations on a concurrent object. After every operation, a thread executes a random number of empty loop iterations (at most 50). This simulates local work and prevents *long runs*, in which a thread would execute bursts of operations on a concurrent object in its local cache. To minimize interference caused by context switching, we assume a uniprogrammed environment, where each thread runs on a separate core (multiprogramming is discussed in Section 6). We pin threads to cores in ascending order, i.e., thread i is pinned to core i. With server-based approaches (SHM-SERVER and MP-SERVER), the server code is executed by thread 0, and other threads execute application code (the server position has a negligible performance impact). In the case of HYBCOMB and CC-SYNCH, all threads run the same code. Unless otherwise stated, the maximum number of requests a thread can combine in HYBCOMB and CC-SYNCH is set to 200 (we analyze this choice later in this section). Every value reported in the graphs is an average over ten one-second runs.

5.3 Microbenchmarks

We first use each of the approaches to implement a simple object, a concurrent counter. Figure 3a shows the throughput of the counter implementations. The approaches that use hardware message passing are clearly faster: MP-SERVER is most efficient in all concurrency levels. Its reaches 4.3x higher throughput than SHM-SERVER, indicating that message passing supported natively is much more efficient than emulation over shared memory. When it comes to combining, HYBCOMB consistently outperforms CC-SYNCH. This is especially pronounced in higher concurrency levels, where HYBCOMB reaches about 2.5x higher throughput. CC-SYNCH and SHM-SERVER have very similar performance, indicating that CC-SYNCH manages to avoid dedicating cores at virtually no performance cost. On the other hand, the difference between MP-SERVER and HYBCOMB is much more visible. We will shortly identify the source of this difference, and explain how it can be minimized.

To give a more complete picture about performance, Figure 3b shows the average request latency observed by application threads. Again, MP-SERVER has by far the lowest latency even in low concurrency levels, indicating that



(a) Throughput (b) Latency (c) Impact of the allowed combining rate

Figure 3. Performance of a concurrent counter implemented using different synchronization techniques

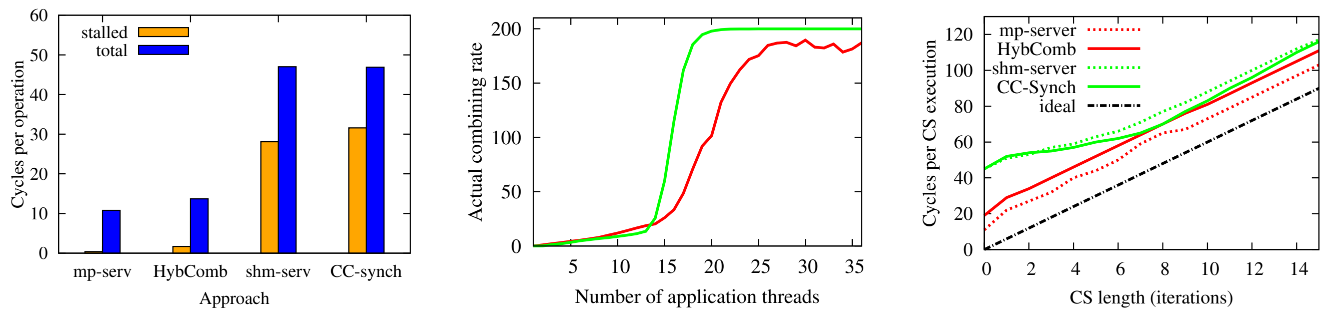
hardware message passing is useful even latency-wise. HYBCOMB also has lower latency than CC-SYNCH and SHM-SERVER, which becomes especially visible as concurrency increases. The only noteworthy exception is single-threaded performance, where CC-SYNCH is better than HYBCOMB. We believe this is mainly because an isolated thread running CC-SYNCH executes only one atomic instruction per operation, whereas HYBCOMB executes three. Since atomic instructions on the TILE-Gx are not executed in the local cache but on memory controllers, this results in higher latency. As concurrency increases, the latency of both CC-SYNCH and HYBCOMB dips at one point before continuing to grow (between 14 and 17, resp. 14 and 24 application threads). This is due to more intensive combining, as we will confirm shortly.

One might question the choice of the maximum allowed combining rate (MAX_OPS). If MAX_OPS is too low, less combining is possible, which negatively affects throughput. On the other hand, increasing it above a certain limit does not increase throughput further, as the cost of combiner switching becomes negligible, but can result in higher latency observed by the combining thread. The optimal value heavily depends on the application needs and anticipated concurrency level. In Figure 3c, we examine how the maximum achievable throughput changes with MAX_OPS. Very high MAX_OPS values provide little benefit in terms of throughput of CC-SYNCH. On the other hand, as we increase MAX_OPS up to 1,000, the throughput of HYBCOMB continues to grow, barely showing signs of saturation. Combining is so fast with HYBCOMB, that the impact of combiner switching is visible even when MAX_OPS is high. This explains the difference between MP-SERVER and HYBCOMB observed in Figure 3a (recall that MAX_OPS is set to 200 there). The throughput of HYBCOMB levels off at about 88 Mops/sec, with MAX_OPS set to 5'000. Therefore, one can achieve nearly as high throughput with HYBCOMB as with MP-SERVER, if willing to trade the throughput increase for sporadic latency "hiccups" for some requests (when the requesting thread becomes a combiner). We have chosen a moderate value of 200 for our experiments, since it already provides the highest possible throughput with CC-SYNCH and decent results with HYBCOMB.

Now we more precisely identify the reason for the observed performance improvement. Figure 4a shows the average number of CPU stalls per operation on the servicing thread under maximum load, as well as the total number of cycles per operation.[4] The advantage of HYBCOMB and MP-SERVER becomes clearer: The servicing thread is virtually never stalled, whereas CPU stalls account for more than 50% of the cycles of the servicing thread in CC-SYNCH and SHM-SERVER. There are no event counters that would provide more fine-grained information on the source of stalls, but we believe they mostly originate from the load-store unit, which has to wait for the cache coherence protocol to fetch data. This confirms the reasoning from Section 3: Cache-coherence related stalls are an important source of overhead, and hardware message passing is helpful in avoiding them.

Figure 4b shows the average combining rate with HYBCOMB and CC-SYNCH. Ideally, we expect it to reach MAX_OPS under high load. At the beginning, the actual combining rate steadily grows, and is approximately equal to the number of threads minus one. This is because a combiner manages to combine one request for all of the other threads. At that point, no thread has started the subsequent operation yet, so the combiner returns. As concurrency grows, more requests arrive at the combiner concurrently. As it takes more time to service them, there is more time for other requests to arrive before the combiner returns, and so forth. This circular effect leads to a sudden sharp increase in the combining rate, which explains the latency dip we observed in Figure 3b. As we can see in Figure 4b, in high concurrency levels CC-SYNCH reaches the desired combining rate, whereas HYBCOMB is slightly below it. This is because registering as a combiner and resetting the n_ops field are not atomic. As explained in Section 4.2, an unfortunate thread interleaving could leave one combiner with no work to do because a new

[4] To be able to use per-core event counters, only in this experiment we modified HYBCOMB and CC-SYNCH to have a fixed combiner for the whole run, which is equivalent to setting $MAX\_OPS = \infty$.



(a) CPU Stalls (b) Combining rate (c) Long CS

Figure 4. Analyzing the performance of the different synchronization techniques

thread would register as a combiner before any request is associated with the current one. However, we can see that this has only a marginal effect on the combining rate in practice: In spite of somewhat lower combining rate, HYBCOMB still has much better performance than CC-SYNCH (Figure 3).

To complete the analysis, we now examine what happens when the CS body is longer. We implement a CS in which the elements of an array are incremented in a loop (one increment per iteration). In Figure 4c, we vary the number of iterations and observe the average CS execution time (the dash-dot line is the time to execute the CS body without synchronization overheads). With MP-SERVER and HYBCOMB, the overhead due to synchronization is constant. The overhead of CC-SYNCH and SHM-SERVER initially decreases as the CS length increases. When the CS is short, their overhead is about 30 cycles higher than with MP-SERVER, which corresponds to the stalled cycles observed in Figure 4a. As the CS gets longer, the RMRs due to thread synchronization get partially overlapped with the CS execution, leading to fewer stalls. Hence, Figure 4c shows that MP-SERVER and HYBCOMB can lead to better performance mainly when CSes are short. At 15 loop iterations, the difference between the best (MP-SERVER) and the worst (SHM-SERVER) performer drops to about 10%, since the time to execute the CS body (which is the same with all of the implementations, if we ignore combiner switching) dominates the entry/exit overhead.

Finally, recall that HYBCOMB uses CAS, but the presented graphs indicate that this does not cause visible performance degradation. This is because, when concurrency is high, threads rarely execute CAS: They mostly send their requests to an active combiner. Indeed, we have measured as few as 0.1 executed CAS per operation (call to *apply_op*) in high concurrency levels. This number is a bit higher when concurrency is not high enough to trigger high combining rates, but even then, there are not more than 0.7 CAS per operation in multithreaded executions. Regarding fairness, we have measured the ratio between the highest and lowest number of operations executed by some thread (so 1 denotes ideal fairness). Across the whole concurrency spectrum, the highest value of this ratio with HYBCOMB is 1.2 and the average is 1.16. Even MP-SERVER, in which all requests are read from a hardware FIFO queue, has a ratio of nearly 1.1, only because some cores are nearer to the server, so they execute slightly more operations. Hence, the use of CAS in HYBCOMB does not impair fairness on this platform.

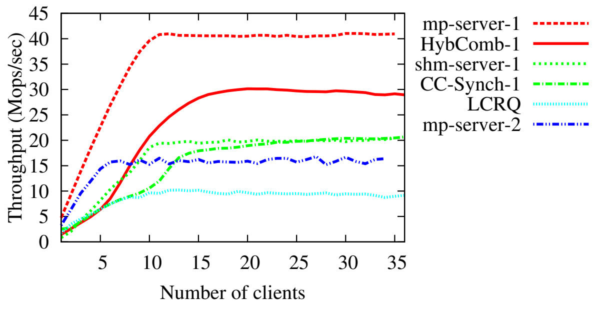
5.4 Queues and Stacks

Because of their ubiquity, concurrent linearizable queues and stacks have been extensively studied and they are typically used to evaluate the performance of universal synchronization constructions [10, 11, 13]. Following this observation, we implement some well-established queues and stacks from the literature and analyze their performance. With these experiments, we study an important use case where CSes are usually short. The implementations store 64-bit values, and are evaluated under balanced load.

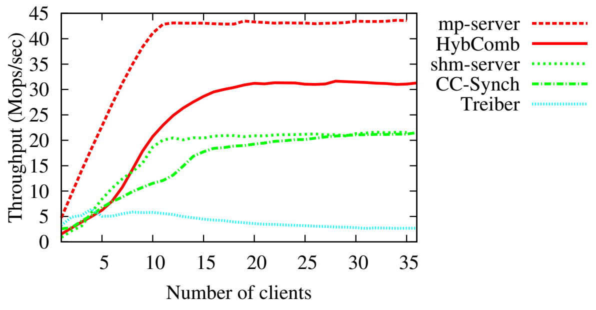
Queues. One of the best-known blocking queues is the fine-grained Michael and Scott queue (MS-Queue) [21]. It is based on a linked list accessed using two CSes, so enqueues and dequeues can take place in parallel. Its performance mostly depends on the way CSes are implemented. We implement MS-Queue using HYBCOMB, CC-SYNCH, and the two server-based approaches (which requires two dedicated servers per queue instance). Besides the two-lock version, we implement the same queue using a single lock. We also test LCRQ [22], a nonblocking queue that takes advantage of the wide spectrum of atomic operations supported by x86 processors. The TILE-Gx supports most of the necessary instructions, so adapting the LCRQ code written in C for x86 was relatively easy.[5]

The queue performance is shown in Figure 5a. The single-lock MS-Queues ("-1" suffix in the legend) perform best. Among them, MP-SERVER and HYBCOMB are most efficient: They obtain respectively up to 2x and 1.5x higher throughput than the third best implementation. LCRQ, as

[5] We made the following modifications: the lacking bitwise test-and-set (BTAS) was replaced with a simple CAS loop; for lack of the 128-bit CAS (CAS2), we modified LCRQ to store 32-bit values, and used a 64-bit CAS.



(a) **Queue.** X-1 – one-lock MS-Queue implemented using approach X; MP-SERVER-2 – two-lock MS-Queue, implemented using MP-SERVER; LCRQ – nonblocking queue as presented in [22]



(b) **Stack.** X – coarse-lock stack implemented using approach X; Treiber – nonblocking stack presented in [28]

Figure 5. Performance of concurrent queues and stacks under balanced load

well as the two-lock MS-Queue[6], level off sooner than the rest, which we now explain in more detail.

One might expect fine-grained locking to always outperform a coarse lock. However, fine-grained locking involves a tradeoff, since the additional synchronization it includes might outweigh the gain that comes from increasing parallelism [13]. Given Tilera's relaxed memory model, the enqueue and dequeue methods of the two-lock queue must be carefully coded if they can run in parallel – memory fences are necessary to ensure queue consistency. On this platform, it turns out that the necessity of inserting fences far outweighs the benefit from fine-grained access. Therefore, a simple sequential queue implemented using MP-SERVER or HYBCOMB yields best results.

In spite of its excellent performance on x86 [22], LCRQ is less efficient on the TILE-Gx. This is primarily because of the way atomic instructions work on this processor. Namely, there are two memory controllers in charge of executing them. This means that two atomic instructions might collide on the memory controller even if they have independent data sets. Because LCRQ executes many atomic instructions per queue operation, such *false serialization* is very frequent, resulting in performance degradation.

Stacks. The stack is known to be hard to parallelize, since both push and pop operations access its top. One way to obviate its seemingly inherent sequential nature is to use the the *elimination* technique [8, 25]: if a push and pop operation are executed concurrently, they can be *eliminated* to avoid accessing the stack. Still, if an operation cannot be eliminated, it has to access the top of the stack. As elimination is orthogonal to the content of this paper, we evaluate the performance of a non-elimination concurrent stack (which, of course, can be used to back up an elimination-based stack).

We evaluate five implementations: a sequential linked-list based stack, turned concurrent using MP-SERVER, HYBCOMB, CC-SYNCH and SHM-SERVER, as well as well-known Treiber's nonblocking stack [28]. Their performance is given in Figure 5b. MP-SERVER and HYBCOMB stacks are again the best performers – and the numbers nearly match those given in Figure 5a for the single-lock MS queue. This is not surprising, as both concurrent objects are represented as linked lists protected by a coarse lock. Treiber stack performance is inferior to that of the blocking implementations, because the head of the stack is accessed using CAS. This causes growing contention as concurrency increases, as most CAS operations repeatedly fail.

5.5 Discussion

One might wonder to what extent our results are processor-specific. To answer this question, we have measured the throughput of a concurrent counter implemented using CC-SYNCH and SHM-SERVER on two single-socket x86 processors: a 10-core Intel Xeon E7-L8867 (without and with Hyperthreading enabled), and a 6-core AMD Opteron 6176. In virtually all of the cases, peak throughput is significantly lower on x86. We have also measured the number of stalls per operation of the servicing thread (as in Figure 4a) and got proportionally larger numbers than on the TILE-Gx. Therefore, we believe MP-SERVER and HYBCOMB would outperform their purely shared-memory counterparts also on x86 hardware, if it provided native message passing support. Moreover, since there are more stalls on x86, the potential performance improvement is higher.

Still, it is noteworthy that we did observe some platform-specific effects. Since the implementation of atomic instructions differs on the TILE-Gx and the x86, algorithms that use them intensively (typically nonblocking ones) may behave differently. This is visible on the example of LCRQ, which has substantially higher throughput on the x86 processors than on the TILE-Gx. Also, because of the different memory consistency model, one-lock MS-Queue outperforms its two-lock counterpart on the TILE-Gx (cf. Figure 5a), in contrast to what we have observed on the Xeon

[6] To avoid clutter, we only present the MP-SERVER implementation of the two-lock queue. The other implementations have inferior performance.

and Opteron. Note, however, that these differences are specific to implementations of a certain concurrent object, a queue in this case. In other words, Figure 5a (showing queue performance) would look different on an x86, but the qualitative advantage of MP-SERVER and HYBCOMB over SHM-SERVER and CC-SYNCH, which is central to this paper, would in all likelihood remain the same.

Finally, the advantage provided by MP-SERVER and HYBCOMB is due to the way hardware message passing is implemented, and more specifically, to the fact that receive operations read from a local buffer, and that send operations are asynchronous. These features are not too specific, and so, we believe they can be easily provided by future implementations of hardware message passing. Note also that HYBCOMB depends a lot on the performance of the fetch-and-add instruction, since every client must execute it on the same variable before sending a request to the current combiner. Fetch-and-add on x86 processors is typically fast and scalable, since it is guaranteed to succeed [22].

6. Additional Considerations

This section discusses some practical aspects of our message-passing approaches.

Oversubscribing and thread migration. The results presented in Section 5 assume a uniprogrammed environment, with at most one thread pinned to a core. This is not an inherent limitation of the hardware message passing approaches. On the TILE-Gx, oversubscribing is easily achieved thanks to the possibility to multiplex the hardware queue of each core (cf. Section 5.1), which means that up to four threads can share a core and still have their exclusive message queue. With both MP-SERVER and HYBCOMB, application threads can freely migrate to another core in between requests, as long as they are able to reserve a hardware queue on that core. Upon making a request, a thread t is only expected to have a valid identifier, corresponding to its current core and hardware queue. As long as t remains pinned to the current core while its request is pending, other threads will be able to reach it using that identifier.

Deadlocks. Bearing in mind the limited capacity of the hardware message queues, another practical issue with message passing is the possibility of deadlocks, if messages back up in the network and block the sender. Obviously, the message queues of MP-SERVER clients or HYBCOMB non-combiner threads cannot overflow since they contain at most one message. Therefore, the servicing thread never blocks when sending a response to a request.

In our experiments, the message queue of a servicing thread cannot overflow, as it contains at most 35 3-word requests at any time, which fits in the message queue. More generally, overflows can happen if the hardware queue is not big enough to keep one request per application thread. In this case, some clients could be blocked when sending a request, but this is not an issue since every such *send* is anyway immediately followed by a blocking *receive*.

7. Related Work

In Section 3, we detailed generic shared-memory constructions for implementing concurrent objects. This section gives an overview of other work studying message passing in the manycore context.

Due to the uncertain future of cache coherence, a great body of recent work studies manycores provided with hardware message passing such as the Tilera [2] and the Intel SCC [16]. It has been shown that message passing can help in achieving good performance in the implementation of transactional memory [12] and key-value stores [7]. In the 90's, Herlihy et al. showed, by simulating MIT's Alewife processor, that message-passing implementations of counting networks and combining trees are more efficient than their shared-memory counterparts [14]. In this paper, we leverage message passing to efficiently implement an arbitrary concurrent object, through universal constructions.

Some recent work also considers hardware augmentations for efficient mutual exclusion: token-based messaging over a dedicated network [3] and a custom instruction set and dedicated cores [27]. Our paper complements these studies by considering an off-the-shelf processor with generic hardware support for message passing, and providing synchronization completely in software.

Finally, similarly to RCL [17], recent work implements message passing over shared memory in the context of concurrent objects [8, 20], because of the explicit control over communication and improved data locality it provides. Our results show that in this case, performance is still limited by the underlying CC protocol, and that hardware message passing can provide a performance improvement.

8. Conclusion

Considering the problem of executing contended critical sections, we studied how hardware message passing can be used for efficient thread synchronization. We proposed two generic constructions tailored to take advantage of hardware message passing: MP-SERVER, a server-based approach, and HYBCOMB, a combiner-based approach. Experiments on Tilera's TILE-Gx processor show that MP-SERVER and HYBCOMB largely outperform their pure-shared-memory counterparts, when used to implement ubiquitous linearizable concurrent objects (counters, queues, stacks).

Our results show that hardware message passing can provide more efficient thread synchronization, and thus, improve the scalability of concurrent code. The hybrid design of HYBCOMB demonstrates that processors providing both CC shared memory and message passing are appealing, as they allow us to take the best of both worlds. However, it also illustrates that significant algorithmic effort can be necessary in order to exploit the resources of a hybrid machine.

Acknowledgments

We would like to thank EcoCloud[7] for providing access to the Tilera processor. Thanks also to Martin Biely, Omid Shahmirzadi and Vasileios Trigonakis for useful comments.

References

[1] Kalray. `http://www.kalray.eu`. Accessed: 15-12-2013.

[2] Tilera. `http://www.tilera.com`. Accessed: 15-12-2013.

[3] J. L. Abellán, J. Fernández, and M. E. Acacio. GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, 2011.

[4] S. Agathos, N. Kallimanis, and V. Dimakopoulos. Speeding up OpenMP tasking. In *Proceedings of the 18th international conference on Parallel Processing*, 2012.

[5] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[7] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, 2011.

[8] I. Calciu, J. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[9] J. Cleary, O. Callanan, M. Purcell, and D. Gregg. Fast asymmetric thread synchronization. *ACM Transactions on Architecture and Code Optimization*, 9(4):27:1–27:22, Jan. 2013.

[10] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, 2011.

[11] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[12] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010.

[14] M. Herlihy, B.-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, Nov. 1995.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[16] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International IEEE Solid-State Circuits Conference Digest of Technical Papers*, 2010.

[17] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[18] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[20] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996.

[22] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[23] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

[24] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, 1999.

[25] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the 7th annual ACM symposium on Parallel algorithms and architectures*, 1995.

[26] D. Sorin, M. Hill, and D. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[27] M. A. Suleman, O. Mutlu, M. Qureshi, and Y. Patt. Accelerating Critical Section Execution with Asymmetric Multicore Architectures. *IEEE Micro*, 30(1):60–70, Jan. 2010.

[28] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[29] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, Apr. 2009.

[7] `http://www.ecocloud.ch`