2011 IEEE International Parallel & Distributed Processing Symposium

GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs

José L. Abellán, Juan Fernández and Manuel E. Acacio
Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia
Facultad de Informática, Campus de Espinardo s/n, 30100 Murcia, Spain
email: {jl.abellan, juanf, meacacio}@ditec.um.es

***Abstract*—Synchronization is of paramount importance to exploit thread-level parallelism on many-core CMPs. In these architectures, synchronization mechanisms usually rely on shared variables to coordinate multithreaded access to shared data structures thus avoiding data dependency conflicts. Lock synchronization is known to be a key limitation to performance and scalability. On the one hand, lock acquisition through busy waiting on shared variables generates additional coherence activity which interferes with applications. On the other hand, lock contention causes serialization which results in performance degradation. This paper proposes and evaluates *GLocks*, a hardware-supported implementation for highly-contended locks in the context of many-core CMPs. *GLocks* use a token-based message-passing protocol over a dedicated network built on state-of-the-art technology. This approach skips the memory hierarchy to provide a non-intrusive, extremely efficient and fair lock implementation with negligible impact on energy consumption or die area.**

A comprehensive comparison against the most efficient shared-memory-based lock implementation for a set of microbenchmarks and real applications quantifies the goodness of *GLocks*. Performance results show an average reduction of 42% and 14% in execution time, an average reduction of 76% and 23% in network traffic, and also an average reduction of 78% and 28% in energy-delay^2 product (ED^{2}P) metric for the full CMP for the microbenchmarks and the real applications, respectively. In light of our performance results, we can conclude that *GLocks* satisfy our initial working hypothesis. *GLocks* minimize cache-coherence network traffic due to lock synchronization which translates into reduced power consumption and execution time.

***Keywords*-Many-core CMP, lock synchronization, global line.**

I. Introduction and Motivation

Multicore architectures (chip-multiprocessors or CMPs) constitute nowadays the best way to take advantage of the increasing number of transistors available in a single die. In particular, they provide higher performance and lower power consumption than more complex unicore architectures. This is due to the fact that these architectures mainly focus on exploiting thread-level parallelism (TLP) rather than instruction-level parallelism (ILP).

While the number of cores currently offered in general-purpose CMPs has already gone above ten (e.g., the 12-core 2-die AMD's Magny-Cours design [1]), the well-known Moore's Law states that soon there will be available on-chip the resources required to integrate dozens of cores or even hundreds of them. As an example, Intel has recently announced the 48-core Single-chip Cloud Computer [2], an experimental research microprocessor that has been developed in the context of the Tera-scale Computing Research Program. CMPs of this kind are commonly referred to as many-core CMPs.

If current trends continue, future many-core CMP architectures will implement the hardware-managed, implicitly-addressed, coherent caches memory model [3]. With this memory model, all on-chip storage is used for private and shared caches that are kept coherent by hardware. Communication between threads is performed by writing to and reading from shared memory. In order to guarantee the integrity of shared data structures, most current systems support synchronization through a combination of hardware (special instructions, such as *LL/SC*, or atomic read-modify-write instructions, such as *test&set*, that operate on shared memory) and software (higher-level mechanisms such as locks or barriers implemented atop the underlying hardware primitives) [4]. In this way, implementations of locks usually rely on shared variables which are atomically updated.

The use of shared variables for lock synchronization has two important implications for performance and scalability in many-core CMPs. First, the cache coherence protocol must come into play in order to maintain the consistency of shared variables across all levels of the memory hierarchy. Coherence activity translates into traffic injection in the interconnection network. As a result, an ever-growing amount of resources may need to be devoted to support lock synchronization as the number of cores in many-core CMPs increases. Moreover, lock acquisition and release operations timing is deeply affected by the performance and scalability of the cache coherence protocol especially under the presence of highly-contended locks. Second, lock contention has long been recognized as a key impediment to performance and scalability since it causes serialization [5]. Consequently, the longer the idle time spent on lock acquisition and release operations, the larger the parallel efficiency reduction.

As an evidence, we show in Figure 1 the potential benefits to performance when lock synchronizations do not involve the cache coherence protocol and have zero latency. To do that, the Raytrace application from the SPLASH-2 benchmark suite [6] is run by using distinct lock implementations

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.87

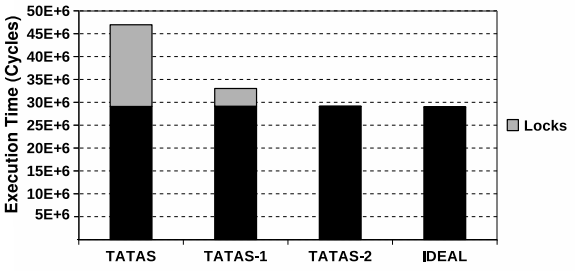


Figure 1. Potential benefits for Raytrace when using *ideal locks*.

(for details about the evaluation see Section IV). In each case, we highlight in gray the fraction of the execution time due to the locks. Shared-memory-based locks use `test-and-test&set` (see TATAS bar in Figure 1). In turn, ideal locks (see IDEAL bar in Figure 1) do not deal with the cache coherence protocol to eliminate any inherited performance or scalability side effect. Besides, lock acquisition and release operations take a single clock cycle each to minimize serialization due to contention. As expected, ideal locks clearly outperform shared-memory-based locks since the lock acquisition and release operations account for a significant fraction of the execution time in Raytrace. However, a post-mortem analysis of Raytrace lock usage reveals that only 2 out of its 34 locks are highly-contended. In this sense, if all the locks other than the highly-contended ones are implemented using regular shared-memory-based locks, a reduction in the execution time similar to that of ideal locks is obtained (see TATAS-1 and TATAS-2 bars[1] in Figure 1). The latter result suggests that only highly-contended locks can truly benefit from a more efficient lock implementation.

In this paper, we present and evaluate a new lock synchronization mechanism aimed at accelerating highly-contended locks. Our proposal, namely *GLocks*, leverages existing global lines technology [7] to deploy a dedicated on-chip global-line-based network. The use of a very simple token-based message-passing protocol over this network provides an extremely efficient implementation for highly-contended locks. To show the benefits derived from *GLocks*, we evaluate the performance of several microbenchmarks and real applications from the SPLASH-2 benchmark suite [6] on a 32-core CMP simulator [8]. Performance results show an average reduction of 42% and 14% in execution time for the microbenchmarks and the real applications, respectively. In addition, they exhibit an average reduction of 76% and 23% in network traffic, for the microbenchmarks and the real applications respectively, given the fact that *GLocks* do not deal with the main data network. This traffic reduction also leads to an average reduction of 78% and 28% in energy-delay^2 product (ED^{2}P) metric for the full CMP, for the microbenchmarks and the real applications, respectively.

[1]TATAS-X means that one (X=1) or two (X=2) of the highly-contended locks have been implemented as ideal locks.

The rest of the paper is organized as follows. Section II discusses some related works on lock synchronization, including background on global lines technology. We detail the architecture, operation, properties and cost of *GLocks* in Section III. Section IV describes our simulation environment and analyzes the obtained performance benefits in terms of reductions in execution time, network traffic and power consumption. Finally, Section V presents our main conclusions and plans for future work.

II. Related Work

The simplest software-based synchronization algorithms rely on atomic read-modify-write instructions, such as `test&set`, `fetch&operation`, `swap` or `compare&swap`, to implement the lock and unlock synchronization primitives [4]. For instance, *Simple Lock* repeatedly tries to acquire the lock by toggling a boolean flag from false to true with a `test&set` instruction. Next, the lock is released by simply toggling the flag back from true to false. The main drawback of this algorithm is the continuous generation of cache-coherence network traffic while busy waiting for lock acquisition. To ameliorate this problem two optimizations, namely `test-and-test&set` and exponential back-off, have been proposed. The former issues standard loads that hit on the local cache while busy waiting for lock acquisition. Hence, the `test&set` is only issued when the lock appears to be free thus reducing cache-coherence network traffic. The latter inserts a delay between consecutive attempts to acquire the lock in order to reduce contention. Anderson [9] found that exponential back-off is the most effective form of delay. Nevertheless, as contention increases these improvements are not enough to guarantee scalability especially for highly-contended locks.

More elaborated algorithms such as *Ticket Lock*, *Array-based Lock* and *MCS Lock* provide more scalable and fair lock implementations at the expense of increased storage cost and higher latency for the low contention case [4]. The *Ticket Lock* algorithm consists of a pair of counters, a *ticket* counter and a *now-serving* counter. To acquire a lock a thread gets its turn by issuing a `fetch&increment` on the ticket counter and then busy waits until the now-serving counter equals its ticket. To release the lock a thread simply increases the now-serving counter. *Array-based Lock* just replaces the now-serving counter by an array of locations. The idea behind *MCS Locks* [10] is similar to that of *Array-based Locks*. An *MCS Lock* builds a distributed queue of waiting threads requesting the lock. In this way, each thread busy waits on a unique, locally-accessible flag rather than competing for a single counter. *MCS Locks* are considered the most efficient software algorithm for lock synchronization [10], [11], [12]. In all three cases, cache-coherence network traffic is reduced because only one thread

actually attempts to obtain the lock when it is released by the previous owner.

In general, simple algorithms tend to be fast under low contention and inefficient when contention is high. In contrast, sophisticated algorithms specifically designed to deal with contention usually incur a non-negligible overhead when there is little contention. For this reason, a number of hybrid approaches have been proposed. *Reactive Lock* [13] is a library-based adaptive approach that chooses the best synchronization algorithm under different levels of contention. This technique switches between *Simple Lock* and *MCS Lock* for the low and high contention cases, respectively. *Smart Lock* [12] uses heuristics and machine learning to choose the most appropriate algorithm following a specific user-defined goal in terms of performance, power consumption or problem-specific criteria.

A completely different software approach that is not based on atomic read-modify-write instructions called *MP-Locks* is presented in [14]. With *MP-Locks* synchronization operations are implemented using message passing, over the main data network, and embedded kernel lock managers. This approach comes into three different flavors, namely centralized, distributed and reactive that differentiate from each other in how the lock managers control lock ownership. A comparison between *MP-Locks* and *MCS Locks* reports significant performance and scalability gains at the expense of increased software complexity and limited portability. A similar idea, proposed in the context of distributed systems, called *Token-based Locks* appears in [15]. In this case, the right to acquire a lock is represented by a token which is unique in the whole system. Threads willing to acquire a lock must wait for token arrival and release the token upon critical section completion.

Hardware support for lock synchronization has also been the target of a number of proposals. Queue-On-Lock-Bit (QOLB) [11] is based on a distributed queue of waiting threads requesting the lock. Unlike *MCS Locks*, in QOLB the queue is implemented entirely in hardware at the cache controller level. The Synchronization-operation Buffer (SB) [16] is a hardware module which augments the memory controller to queue and manage lock operations issued by the threads. QOLB reports non-negligible performance gains when compared to *MCS Locks*. In general, all of the hardware-supported solutions require modifications at some level of the memory hierarchy. Contrarily, our approach completely decouples lock synchronization from any kind of memory-related activity.

Differently from previous proposals, our approach is based on the use of Global lines (*G-lines* from now on). *G-lines* have already been successfully integrated in a silicon substrate in order to enable speed-of-light point-to-point communications. Chang *et al.* [7] and Jose *et al.* [17] showed early point-to-point circuits allowing transmission-line, wave-like velocity for 10mm of interconnect. Nonetheless, this initial implementation suffers from significant power consumption and die area overheads.

A great effort has been devoted to overcome such limitations. For instance, Ito *et al.* [18] extend *G-lines* to support broadcast, multi-drop and bidirectional transmissions. This contribution enables both low-latency and multi-drop ability on a transmission line with low-power dissipation. However, their results still exhibit several integration density issues. Additionally, Ho *et al.* [19] and Mensink *et al.* [20] have shown that a capacitive feedforward method of global interconnect reduces both power consumption and die area overheads. In particular, they achieve nearly single-cycle delay for long wires with voltage-mode signaling. As a result, every *G-line* is basically a shared wire that broadcasts 1-bit messages (signals from now on) across one dimension of the chip in a single clock cycle. A practical use of *G-lines* is presented by Khrisna *et al.* [21] in the context of networks-on-chip (NoC). Khrishna *et al.* leverage *G-lines* using multi-drop connectivity and S-CSMA collision detection to enhance a flow control mechanism (EVC) in terms of latency and power consumption. In particular, these *G-lines* are to broadcast the control signals of EVC in order to communicate the availability of free buffers and virtual channels much more accurately (original EVC uses thresholds to conservatively communicate available resources). Furthermore, the authors employ the S-CSMA technique to calculate how many virtual channels or free buffers are demanded at any time in order to grant requests accordingly. As another example of application of the *G-lines*, we developed in [22] an efficient hardware-based barrier synchronization mechanism that uses *G-lines* and the S-CSMA technique to provide low-latency, power-efficient barrier synchronizations in many-core CMPs. The *GLocks* architecture proposed in this work leverages this technology to deploy a dedicated *G-line*-based network in order to efficiently perform lock synchronizations. Nonetheless, unlike [21] or [22], our proposal does not require the S-CSMA technique, thus saving power and area.

III. The *GLocks* Mechanism

In this section, we present our proposal to build an efficient synchronization mechanism for highly-contended locks in many-core CMPs. We start by describing the *G-line*-based network that our proposal entails. As a case study, we choose a CMP with a 2D-mesh data interconnection network, although our proposal is not restricted to this topology. Next, we show how the *GLocks* mechanism would operate. After that, we describe the interface for programmers and provide details about the implementation of the *G-line* controllers required by our proposal. Finally, we analyze the implementation costs and propose how our mechanism may be extended to tackle with even hundreds of cores.

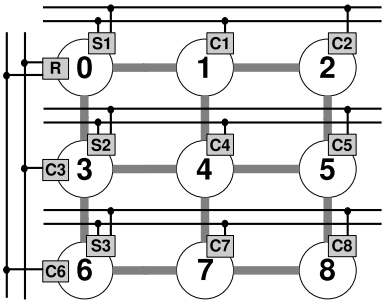


Figure 2. *GLocks* architecture for a 9-core CMP with a 2D-mesh network.

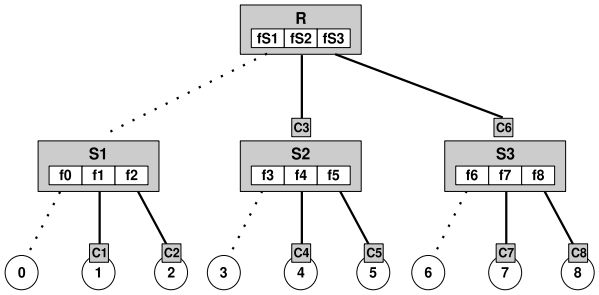


Figure 3. Logical view of the *G-line*-based network for a 9-core CMP with a 2D-mesh network.

A. G-line-based Network

The *GLocks* mechanism proposed in this work relies on a *G-line*-based network as can be observed in the example in Figure 2. For simplicity, we concentrate on a version of the proposed network providing support for one lock. As can be observed, the *G-line*-based network is made up of two kind of components. *G-lines* (horizontal and vertical finer black lines), that are used to transmit the signals required by the synchronization protocol; and *controllers* (R, Sx and Cx), that actually implement the synchronization protocol.

As discussed in Section II, every *G-line* is a wire that enables the transmission of 1 bit of information across one dimension of the chip in a single cycle. In this way, the *G-line*-based network employs one *G-line* per transmitter and lock. Every *G-line* will be used to request the associated lock and grant lock acquisitions. In this way, for any 2D-mesh layout the total number of *G-lines* per lock that would be needed is equal to $C - 1$, where C is the number of cores of the CMP (e.g. 8 *G-lines* for the 9-core CMP shown in Figure 2). It is worth noting that our proposal is aimed at providing this kind of hardware support just for a very limited number of locks, which makes that the total amount of *G-lines* that would be required keep lower than, for example, the 168 used in [21]. Since low die area overhead is reported for the latter, we can affirm that our implementation introduces a negligible overhead.

In addition to the *G-lines*, our proposal also incorporates a set of *controllers*. In particular, we distinguish two types of controllers: the *local controllers* (Cx in Figure 2) and the *lock managers* (R and Sx in Figure 2). The *local controllers* send and receive signals to and from their corresponding *lock managers* through their dedicated *G-lines* (e.g. $C1$ sends and receives signals to/from $S1$). The exception is when the *local controller* is located in the same core as its associated *lock manager*. In this case, the functionality of the *local controller* is encapsulated in the *lock manager*, and communication is performed locally by means of a flag. For example, $S1$ monitors not only signals from *local controllers* 1 and 2 ($C1$, $C2$) through their corresponding *G-lines*, but also from the local core through an internal flag (for clarity, this flag is not shown in Figure 2).

The *lock managers* control lock ownership by monitoring signals from either *G-lines* (remote cores) or the flags (local core). Besides, *lock managers* are divided into two groups: primary and secondary *lock managers*. Secondary *lock managers* (Sx) are responsible for monitoring signals from their corresponding *local controllers*, whereas the primary *lock manager* (R) is responsible for monitoring signals from the secondary ones. Primary and secondary *lock managers* communicate each other by means of the vertical *G-lines* shown in Figure 2.

Finally, to have a clear understanding of our proposal, we represent the architecture described above as the hierarchy shown in Figure 3. In particular, the *G-line*-based network that our proposal is based on can be represented as a three-level hierarchy. The root of the hierarchy is the primary *lock manager*. The secondary *lock managers* would be located at the intermediate nodes. Finally, the leaves of the hierarchy would be the processor cores (with the *local controllers*). All elements are connected using *G-lines* (continuous lines) or locally by means of an internal flag (dashed lines). The flags (*fx* and *fSx*) store the signals sent by the controllers to the corresponding *lock manager* (primary and secondary). In this way, we need flags not only to store the signals sent between Sx and the *local controllers* (one flag per Cx controller: $f1$ for $C1$, $f2$ for $C2$, etc.), but also to store the signals transmitted between R and Sx (one flag per Sx controller: $fS1$ for $S1$, $fS2$ for $S2$, etc.).

B. Synchronization Protocol

The synchronization protocol implemented on top of the *G-line*-based network previously described is based on the exchange of 1-bit messages (signals) between the *local controllers* and the *lock managers*. More specifically, the protocol uses three types of signals to perform a lock synchronization. The REQ and REL signals, which are sent from the *local controllers* to their corresponding *lock manager* to ask for the lock and to release the lock, respectively; and the TOKEN signal which is sent from a *lock manager* to a particular *local controller* to grant access to a lock. In addition, these signals are also transmitted between primary and secondary *lock managers* in a lock synchronization. In

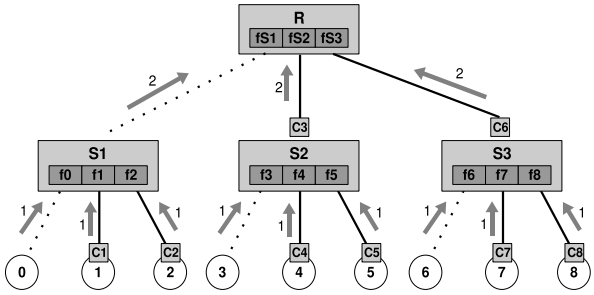
particular, the secondary *lock managers* ask for the lock by sending the REQ signal to the primary *lock manager* and receive authorization from the latter through the TOKEN signal. Similarly, after the lock is released, a secondary *lock manager* notifies the primary one by means of the REL signal.

Lock managers (both the primary and secondary ones) use a round-robin strategy to grant the lock among those processor cores which are competing for becoming the next owner. Let's assume that all of the cores in Figure 3 send the REQ signal to their corresponding secondary *lock manager* at the same time. In this case, the TOKEN signal granting the lock would be received by Core0 first; then, once Core0 has released the lock, Core1 would become the next holder; and so on, until Core8 is reached. Next, the process would start again from Core0 if there are additional pending lock requests. Since the *GLocks* mechanism is aimed at accelerating highly-contended locks we do not expect that the election of the strategy to grant the lock in these situations will have an impact on performance. However, this is a key design point to ensure the fairness expected from a lock implementation [4]. The latter is the reason why we use the round-robin strategy.

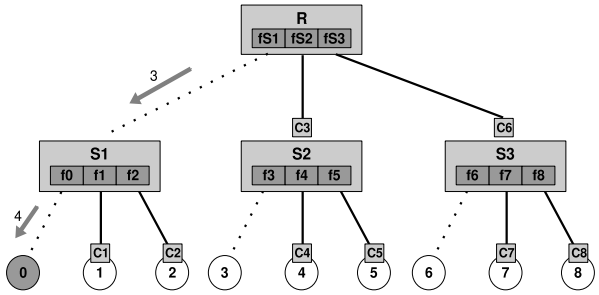
As an example of how the synchronization protocol works, Figure 4 presents the case where the 9 cores of the CMP depicted in Figure 2 try to get access to the lock at the same time. To clarify the explanation, the arrows in the figure mark the sense of the transmissions. Moreover, each arrow is labeled with the cycle in which communication occurs, starting with cycle 1. Finally, we highlight with dark grey the flags that are written and the core that acquires the lock in each case.

At cycle 0, all cores try to get the lock (see Figure 4(a)). To do this, every *local controller* (Cx in the figure) sends the REQ signal at cycle 1 to the corresponding secondary *lock manager* (Sx in the figure). As a result, all fx flags would be written, and each Cx would be busy-waiting until the TOKEN signal is received. At cycle 2, once each Sx detects that at least one of its fi flags has been written, REQ signals towards the primary *lock manager* (R in the figure) are sent in order to write the corresponding fSx flags. At this moment, R must make a decision about the secondary *lock manager* that will be granted the lock ownership. This process is shown in Figure 4(b). In this case, R would choose $S1$ by following the round-robin scheduling policy already discussed and would send the TOKEN signal at cycle 3. At cycle 4 and based on the round-robin policy, $S1$ chooses Core0 and sends the TOKEN signal granting access to the lock.

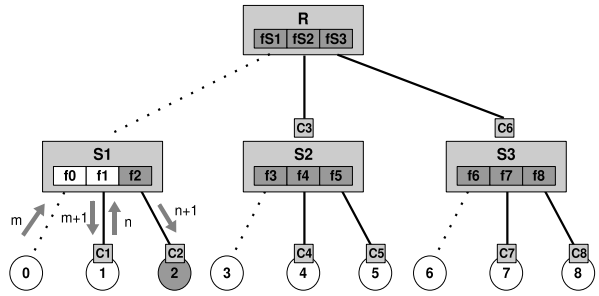
Figure 4(c) shows the scenario in which an Sx can grant the lock ownership without involving any additional notifications to R. More specifically, once Core0 releases the lock at cycle *m*, its controller sends the REL signal (by writing to the local $f0$ flag, as we mentioned) to $S1$. Next,



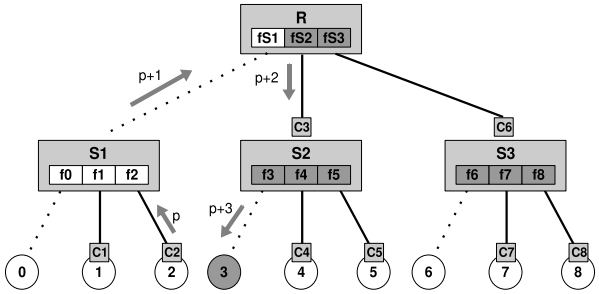
(a) All cores request the lock at the same time (cycle 1).



(b) Lock is granted to Core0 (cycle 4).



(c) Core0 releases the lock (cycle *m*) and *S1* designates Core1 to be the next lock holder (cycle *m+1*).



(d) Core2 releases the lock (cycle *p*) and *S2* designates Core3 to be the next lock holder (cycle *p+3*).

Figure 4. Example of lock synchronization under the *GLocks* mechanism.

at cycle $m+1$, $S1$ grants the lock ownership (by means of the TOKEN signal) to the next core by following the round-robin policy from the active fx flags. In this case, Core1 becomes the new lock holder. In the same way, Core2 would be granted the lock in cycle $n+1$ ($m < n$). Finally, in Figure 4(d) we illustrate the scenario when an S finishes its scheduling because either it has reached the last active f or there are no more pending local requests for the lock. In this case, S must send the REL signal towards R, which will choose another available Sj *lock manager* from those that activated the fSx flags. In the figure, $S1$ sends the REL signal to R at cycle $p+1$ ($n < p$), which following the round-robin policy grants the lock to $S2$. Finally, $S2$ sends the TOKEN signal giving access to the lock to Core3 at cycle $p+3$.

C. Programmability Issues

The *GLocks* mechanism proposed in this work is intended to be used by programmers in a transparent way. For that, as shown in Figure 5, we propose to provide special library-level lock and unlock methods (*GL_Lock* and *GL_Unlock* in the figure) encapsulating the functionality of the *GLocks* mechanism and that could be used in parallel applications to deal with contended locks. In more depth, the interface with the *GLocks* mechanism is accomplished through a couple of flags per lock added to each processor core. The `lock_req` flag is used to request the lock and wait for lock acquisition. As a result of the activation of the `lock_req` flag by a processor core, the corresponding fx flag is activated by the *local controller* and the synchronization protocol explained in the previous section would be invoked. Once the lock is granted, the corresponding `lock_req` flag is reset by the *local controller*. Similarly, a lock release is notified by setting the `lock_rel` flag, which in turn deactivates the corresponding fx flag. Finally, the `lock_rel` flag is also reset after the *local controller* has unset the fx flag. Note that all pairs of flags (one per lock) could be grouped in each core using one special lock register.

As pointed out along this paper, our *GLocks* mechanism is aimed at accelerating highly-contended locks. Obviously, the programmer is responsible for identifying locks of this kind and using the `GL_Lock` and `GL_Unlock` methods previously described for them. In the literature, there have been proposed several heuristics to detect contended locks in those cases in which it could be a tedious or difficult task. As an example, Tallent *et al.* [5] have recently proposed strategies for gaining insight into performance losses due to lock contention. Their goal was to understand where a parallel program needed improving.

D. G-line Controllers Implementation

In this section, we take a closer look at the implementation of the *G-line* controllers (see Figure 6). As we can see, there are three automata corresponding to each of the three

```
GL_Lock() {
   asm {
        # Arrival at the CS: set lock_req
        mov 1, lock_req

        # Busy-wait until lock_req is reset
        loop:
                bnz lock_req, loop
   }
}

GL_Unlock() {
   asm {
        # Release lock: set lock_rel
        mov 1, lock_rel
   }
}
```

Figure 5. Encapsulating the *GLocks* functionality into the lock/unlock library-level methods.

kind of controllers aforementioned: primary and secondary *lock managers*, and *local controllers*. Furthermore, over each transition, we also depict the event that motivates the transition to the next state, and the action that may produce a new event. It follows the pattern: [EVENT] / [ACTION]. In more depth, we distinguish the following events and actions:

- A core writes the registers *lock_req* or *lock_rel*: e.g. *Core(lock_req:=1)*. Notice that we use := to assign a value, and = to compare two values.
- A *G-line* controller writes a *G-line*: *AglineB:=SIGNAL*, where A corresponds to the writer controller and B refers to the receiver controller. Then, A and B could be: LX for a *local controller*; and P and SX stem from primary and secondary *lock manager*, respectively (X is also named as Y or Z to specify that the signal must come from a different controller). Besides, SIGNAL identifies the type of signal (REQ, TOKEN and REL) transmitted across the *G-line*. To specify that there is not a signal in a *G-line*, we use *Gline=off*. Additionally, we distinguish between REQ or REL signals (e.g. see the *Scheduling* state of the *Secondary Lock Manager* automaton in Figure 6) issued from a particular *local controller*, by internally checking the associate *fX* flag. Then, if *fX* is equal to zero the signal is treated as REQ, and REL in the opposite case.
- Updating of registers: *fX:=1*, *fSX:=1* and *lock_ownership:=TRUE*, where X identifies the flag associated to the controller which sends the signal (REQ or REL). Note that we use the *lock_ownership* register to define when a secondary *lock manager* will be granted the lock ownership. In addition, we assume that all registers are initialized to zero or false.
- Round-robin strategy execution to grant the lock: *RoundRobin()=V*, where V refers to NULL if the corresponding *lock manager* has finished its scheduling

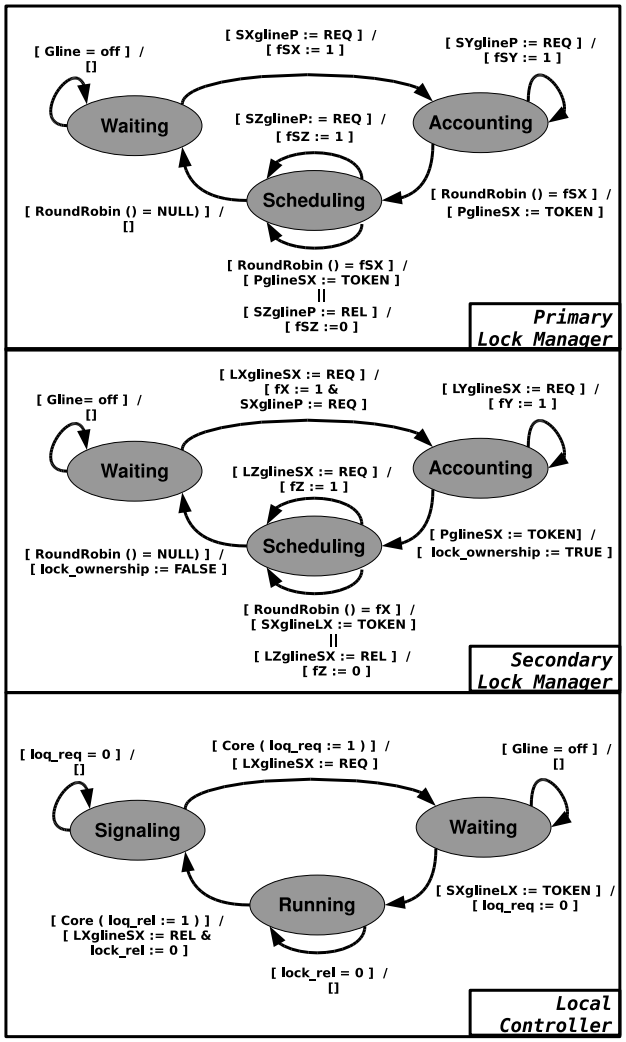


Figure 6. Finite state automata that implement the *G-line* controllers.

because either it has reached the last flag (*fX* or *fSX*), or there are no more pending local requests for the lock, as mentioned at the end of Section III-B. In any other case, V corresponds to the specific flag (*fX* or *fSX*) associated with the next holder controller.

- Finally, no event or action: *[ ]*.

Finally, for the sake of clarity, we omit in the automata when communication is performed locally by means of a flag (see Section III-A). Then, we suppose that all controllers communicate each other by means of signals through *G-lines* to carry out the synchronization protocol.

E. Implementation Costs for GLocks

In this section, we discuss the costs that the *GLocks* mechanism would entail (see Table I). First of all is the number of *G-lines* that must be used to configure the special network. As already commented on, the *G-line*-based network deploys separate sets of *G-lines* per lock. In particular, each lock needs a set of $C-1$ *G-lines* being C the total number of cores. In addition, lock synchronization is achieved using a set of controllers, which includes one primary *lock manager*, $\sqrt{C}$ secondary *lock managers* and $C-1$ *local controllers*[2]. Each of these controllers would implement the simple synchronization protocol described in Section III-B. Primary and secondary controllers would use a set of C and $\sqrt{C}$ flags per lock respectively, whereas the *local controllers* would activate the *G-line* associated to the lock requested by the processor core. Since the number of highly-contended locks is very small in real applications (up to 2 in the applications evaluated in this work), we expect that providing a very limited number of *GLocks* will be enough for all highly-contended locks in most cases. Considering the results reported in [21] that show very small area implications for a 168-*G-line* network, we can affirm that our implementation introduces a negligible overhead.

Table I
HW/SW COST OF *GLocks* FOR A 2D-MESH CMP LAYOUT.

G-lines	$C-1$
Primary *Lock Managers*	1
Secondary *Lock Managers*	$\sqrt{C}$
Local controllers	$C-1$
fSx Flags	$\sqrt{C}$
fx Flags	C
Lock Acquire (worst case)	4 cycles
Lock Acquire (best case)	2 cycles
Lock Release	1 cycle

Finally, differently from shared-memory-based locks, our proposal neither consumes space in memory and local caches with synchronization information nor involves the cache coherence protocol. In this way, the *GLocks* mechanism would avoid the significant amount of traffic that shared-memory-based locks would introduce in the main data network in high-contention situations. This would also translate into important energy savings for the interconnection network, as we will show in Section IV-D.

F. Scalability Issues

Scalable lock synchronizations become a crucial issue when dealing with an ever-increasing number of cores in many-core CMPs. In this way, our *GLocks* architecture has been devised to also meet scalability. As in [21], we assume in this work that every *G-line* can support up to six transmitters and one receiver, resulting in a CMP configuration with up to 7×7 cores. We would like to point out that our *GLocks* mechanism is not only restricted to that CMP layout and could be easily extended to support a higher number of cores. In particular, we could extend the scalability of *GLocks* by using either longer latency *G-lines*, or different groups of *G-line*-based networks linked together through additional *G-lines*.

[2]Remember that we are assuming a 2D-mesh physical layout for the CMP.

Table II
CMP BASELINE CONFIGURATION.

Number of cores	32
Core	3GHz, in-order 2-way model
Cache line size	64 Bytes
L1 I/D-Cache	32KB, 4-way, 2 cycles
L2 Cache (per core)	256KB, 4-way, 12+4 cycles
Memory access time	400 cycles
Network configuration	2D-mesh
Network bandwidth	75 GB/s
Link width	75 bytes

IV. EVALUATION

In this section we give details of our experimental methodology and performance results. We describe the simulation environment in Section IV-A, the set of microbenchmarks and real applications we have used in Section IV-B, and the lock implementation the *GLocks* mechanism is compared with in Section IV-C. Finally, the performance results are analyzed in Section IV-D in terms of execution time, network traffic and power consumption.

A. Testbed

In order to support *GLocks*, the Sim-PowerCMP [8] performance simulator has been extended. Sim-PowerCMP is a detailed architecture-level power-performance simulation tool that simulates tiled-CMP architectures with a shared L2 cache on-chip and a MESI directory-based cache coherence protocol. In more depth, Sim-PowerCMP is based on a Linux x86 port of RSIM [23] and models a CMP architecture consisting of arrays of replicated tiles connected over an on-chip network. Each tile contains a processing core with primary caches (both instruction and data caches), a slice of the L2 cache, and a connection to the on-chip network. Sim-PowerCMP estimates power consumption by implementing already proposed and validated power models for both dynamic power (from Wattch, CACTI) and leakage power (from HotLeakage) of each processing core (including the L1 caches), the shared multi-bank L2 cache, as well as the interconnection network (from Orion) [8]. Table II summarizes the values of the main configurable parameters assumed in this work. As it can be seen, we have simulated a 32-core CMP with an aggressive 2D-mesh network.

B. Benchmarks

To evaluate the performance benefits derived from *GLocks*, five microbenchmarks and three scientific applications are used. On the one hand, the microbenchmarks exhibit different highly-contended access patterns to shared data that can be commonly found in parallel applications. Single Counter (SCTR) consists of a counter (fits in a cache line), protected by a single lock, that is incremented by all threads in a loop. Multiple Counter (MCTR) is made up of an array of counters (residing in different cache lines), protected by a single lock, where each thread increments a different counter of the array in a loop. Doubly-Linked list (DBLL) builds a doubly-linked list, protected by a single lock, where threads dequeue elements from the head of the list and enqueue them into the tail of the list afterwards. Producer Consumer (PRCO) consists of a shared FIFO (bounded) array, protected by a single lock, that is initially empty. Half the threads enqueue items into the FIFO that are consumed by the other half of threads. Producers have to wait for free slots in the FIFO whereas consumers have to wait for data to consume before iterating the critical section code. Affinity Counter (ACTR) uses two locks that protect two counters accessed consecutively by all threads. For each iteration, all threads acquire the first lock to update the first counter, barrier synchronizes them, and then the second lock is acquired to modify the second counter. To implement these microbenchmarks we follow a methodology similar to the one used in [24], [25]. On the other hand, regarding real applications, we have considered two programs belonging to the SPLASH-2 benchmark suite [6]: Ocean studies large-scale ocean movements based on eddy and boundary currents; and Raytrace renders a three-dimensional scene using a ray tracing method. Finally, QSort is a well-known sorting algorithm that we implement on a given array of integers. These applications were chosen since they present a significant lock synchronization overhead due to the existence of highly-contended locks[3]. In fact, these locks are accessed following similar patterns to those of the microbenchmarks. We summarize the characteristics of the microbenchmarks and applications used in this work in Table III. For each of them we account for the input size, the total number of different locks, the number of these locks that are highly-contended (H-C Locks), and point out the highly-contended lock access patterns in terms of the microbenchmarks they are similar to.

All experimental results reported in this work are for the parallel phase of all of the benchmarks previously described. Finally, only contended locks are implemented using the *GLocks* mechanism. For the rest of locks, the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization is used. This includes the locks used in the applications' library of our simulator to implement barriers. Apart from not being application-level, these locks do not exhibit high contention levels since our simulator provides applications with an efficient tree barrier implementation (up to two threads requesting every lock). In this way, barriers are not affected by our proposal.

To determine the contention of locks, we performed a post-mortem analysis of the benchmarks under study where locks use the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization. Every time a core tries to acquire a lock, we register the number of concurrent

[3]In this work, highly-contended locks are those locks accessed by all threads simultaneously or very close in time.

Table III
CONFIGURATION OF THE BENCHMARKS AND LOCK-RELATED CHARACTERISTICS.

Benchmark	Input Size	Locks	H-C Locks	Access Pattern
SCTR	1,000 iterations	1	1	-
MCTR	1,000 iterations	1	1	-
DBLL	1,000 iterations	1	1	-
PRCO	1,000 iterations	1	1	-
ACTR	1,000 iterations	2	2	-
RAYTR	teapot	34	2	SCTR
OCEAN	258x258 ocean	3	1	SCTR
QSORT	16384 elements	1	1	PRCO

requesters (group of acquiring cores or grAC ranging from 1 to 32) on a cycle-by-cycle basis until the lock is granted to the core. In this way, we can precisely compute each lock's contention rate as the number of cycles where the number of concurrent requesters is equal to each grAC divided by the total amount of cycles where the number of concurrent requesters belongs to the range [1..32]. That is, the lock's contention rate (*LCR*) of a particular lock (*Lock*) for each grAC ($i \in$ [1..32]) would be defined by equation 1.

$$LCR_{grAC_i} = \frac{Cycles(Lock, grAC_i)}{\sum_{g=1}^{32} Cycles(Lock, grAC_g)} \quad (1)$$

In Figure 7, the lock's contention rate for all of the benchmarks (x-axis) is shown. In particular, we show the lock's contention rate (y-axis) for all of the possible values of grAC (z-axis). Moreover, we decompose the results for each benchmark on a per-lock basis[4]. To do that, we assume that equation 2 is satisfied and redefine equation 1 as equation 3. That is, every lock's contention rate has also been estimated depending on the amount of clock cycles it uses. From this, we can easily identify in Figure 7 those locks that present high contention, and those that although exhibiting high contention are executed during a negligible amount of clock cycles. Due to their very low impact on execution time, the latter kind of locks would be implemented by using the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization.

$$LCR_{benchmark} = \sum_{i=1}^{Locks} \sum_{j=1}^{32} L_i CR_{grAC_j} = 1 \quad (2)$$

$$L_i CR_{grAC_j} = \frac{Cycles(Lock_i, grAC_j)}{\sum_{l=1}^{Locks} \sum_{g=1}^{32} Cycles(Lock_l, grAC_g)} \quad (3)$$

As expected, the microbenchmarks exhibit a very high lock's contention rate when grAC is close to the total number of cores. The exception is the ACTR microbenchmark which presents a moderate homogeneous level of contention across all the grAC range. This is mainly due to the barrier synchronization interleaved between the two lock acquisition operations. The real applications also report a behavior similar to that of the ACTR microbenchmark. In this case the reason is their much coarser granularity which spreads the acquire operations throughout the parallel phase. Finally, it is worth noting that OCEAN and RAYTRACE just have one and two highly-contended locks, respectively.

[4]Although Raytrace has 34 locks, we only include the results for the two most highly-contended locks (RAYTR-L1 and RAYTR-L2) and aggregate the rest (RAYTR-LR).

C. Lock Implementations

To fairly quantify the benefits of our *GLocks* mechanism, we consider the case that highly-contended locks found in the benchmarks previously described are implemented by using *MCS Locks*. As explained in Section II, *MCS Locks* are one of the most efficient software algorithms for lock synchronization. In particular, *MCS Locks* gracefully manage high-contention situations by having a distributed queue of waiting lock requesters. On the other hand, for the rest of locks (non-contended ones), we employ the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization due to it has been shown to lead to lower latencies when threads try to acquire a lock without competition. Finally, since the number of highly-contended locks is commonly very small in real applications (up to 2 in the applications evaluated in this work), we assume that two *GLocks* are provided at hardware level.

D. Performance Results

The evaluation of the *GLocks* mechanism proposed in this work has been carried out taking into account the execution times achieved for the benchmarks shown in Table III, as well as the amount of traffic in the interconnect and the energy-delay^2 product (ED^{2}P) metric for the full CMP.

1) Execution Time: Figure 8 shows the execution times that are obtained for the set of benchmarks under study when either *GLocks* or *MCS Locks* are employed for the highly-contended locks (GL bars and MCS bars respectively). In particular, execution times have been normalized with respect to those obtained when *MCS Locks* are used. Additionally, each bar shows the fraction of the execution time due to lock and barrier synchronizations (*Lock* and *Barrier* categories respectively), memory accesses (*Memory*

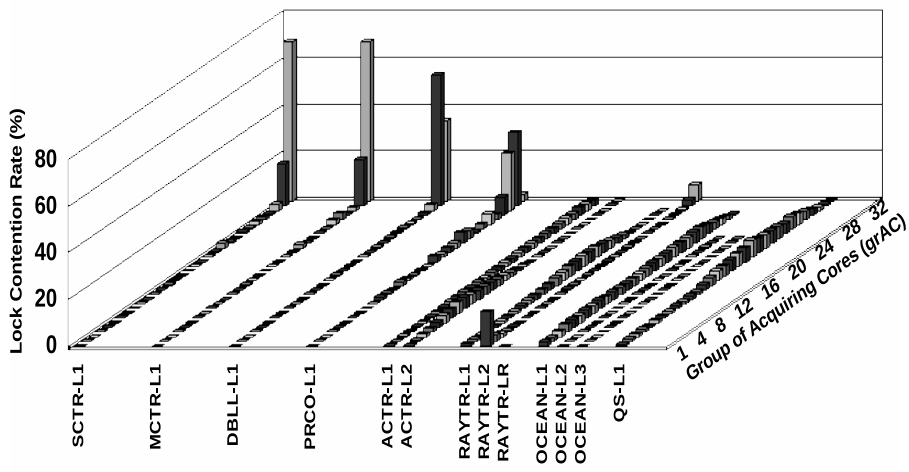


Figure 7. Locks' Contention Rate.

category) and computation (*Busy* category). Finally, average execution times are shown in separate bars for the microbenchmarks (*AvgM*) and applications (*AvgA*).

Regarding the microbenchmarks, we can observe that our proposal presents an average reduction of 42% in execution time (see *AvgM*). The exact extent of the reduction in each case depends on both: the number of highly-contended locks that each microbenchmark has (see Table III), and also the contention rates exhibited by each lock (see Figure 7). In particular, our proposal is applied in SCTR, MCTR, DBLL and PRCO to its single contented lock, resulting in reductions of 33%, 39%, 34%, 25% in execution time, respectively. On the other hand, two contended locks are found in the ACTR microbenchmark, which increases the benefits of our proposal (reductions of 81% are obtained). This high reduction is also explained since ACTR presents a much lower contention rate. In particular, in Figure 7 we can observe that SCTR, MCTR, DBLL and PRCO present a contention rate close to 80% when considering grACs higher than 20 cores. In contrast, ACTR presents an aggregate contention of only 20% for the same grACs. As we mentioned, *MCS Locks* become inefficient for the low contention case, which accentuates even more the differences between *MCS Locks* and our *GLocks* proposal.

A more in depth analysis reveals that the former reductions come from two kind of effects that the *GLocks* mechanism has. First, the time taken to acquire and release the lock is drastically reduced as derived from the improvements shown in the *Lock* category. And second, the fact that our proposal removes from the main data network all extra coherence traffic that a shared-memory-based lock implementation would introduce, also has effect on the *Barrier* category for the ACTR microbenchmark.

On other hand, the fraction of the execution time that lock synchronization consumes is lower when real applications are considered. In these cases, most of the time is spent

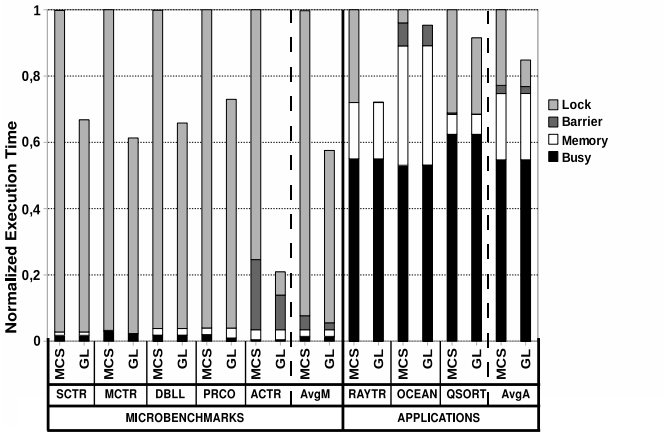


Figure 8. Normalized Execution Time.

on computations and memory accesses (*Busy* and *Memory* categories). This explains the lower reductions in execution time observed for Raytrace, Ocean and QSort (14% on average). Moreover, since QSort presents higher contention rates than Raytrace (aggregate contentions of 60% and 29%, respectively, for grACs higher than 20 cores), the *MCS Locks* become more efficient which translates into lower performance differences between *MCS Locks* and the *GLocks* mechanism.

To conclude this section, we also show in Table IV speedup results for the real applications (Raytrace, Ocean and QSort) when scaling the number of cores parameter with the values 4, 8, 16 and 32. Moreover, we use two different lock implementations for the high contention case: *MCS Locks* (MCS) and our *GLocks* mechanism (GL). From the results shown in Table IV, we can extract two important observations. First, all of the benchmarks scale as the number of cores is increased. Second, the exact extent of speedups

Table IV
SPEEDUPS FOR THE REAL APPLICATIONS.

Benchmark	Lock Version	4	8	16	32
RAYTR	MCS	3.91	7.53	13.61	20.69
	GL	3.93	7.97	15.67	28.78
OCEAN	MCS	3.7	7.12	13.48	23.62
	GL	3.8	7.32	13.93	25.66
QSORT	MCS	3.67	6.49	9.68	11.38
	GL	3.69	6.55	9.92	12.4

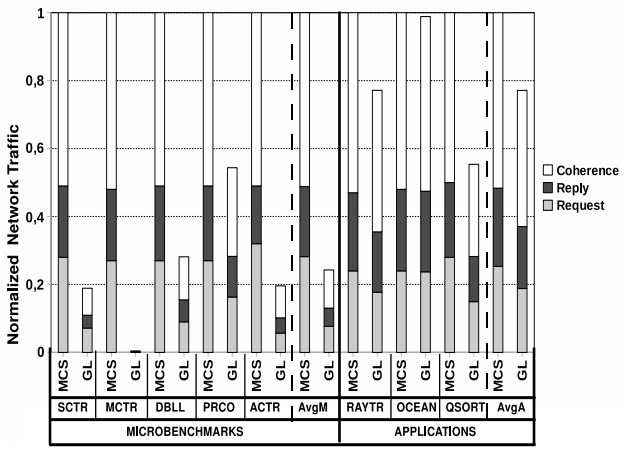


Figure 9. Normalized Network Traffic.

depends on the efficiency of the lock implementation we are using. In this way, higher speedups are obtained when employing our *GLocks* mechanism which are even very close to ideal speedups in the case of Raytrace.

2) Network Traffic: Our proposal does not generate any coherence messages on the main data network when performing lock synchronizations. At the end, this translates into significant reductions in terms of network traffic. Figure 9 shows the total network traffic across the main data network. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to the *MCS* case. Each bar is broken down into three categories: *Coherence* corresponds to the messages generated by the cache coherence protocol (e.g. invalidations and Cache-to-Cache transfers); *Request* comprehends messages generated when load and store instructions miss in cache and must access a remote directory; and finally, *Reply* involves the messages with data.

For the microbenchmarks, important reductions in network traffic are achieved (76% on average). In general, these reductions are directly related to the extents of the improvements in execution time previously reported. Moreover, since the simulated L2 cache is shared among the different processing cores, but it is physically distributed between them (see Section IV-A), some accesses to the L2 cache will be sent to the local slice while the rest will

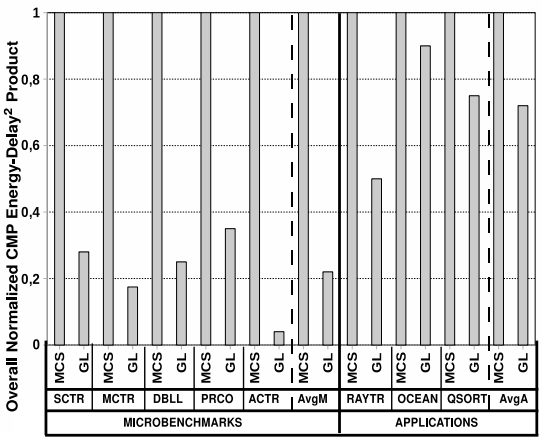


Figure 10. Normalized energy-delay^2 product (ED^{2}P) metric for the full CMP.

be serviced by remote slices. This will also affect to lock acquisition and release operations timings. In contrast, since our *GLocks* implementation skips the memory hierarchy we have not obtained such negative impact on network traffic. In particular, SCTR, MCTR, DBLL and PRCO show reductions of 81%, 99%, 72% and 46%, respectively. This is due to the fact that almost all network traffic of these microbenchmarks is due to lock synchronizations. The exception is ACTR, where the barrier used in between the two phases also generates network traffic. However, since the barrier time is approximately 20% of the lock time (see *Barrier* and *Lock* categories in Figure 8), a reduction of 80% in network traffic is obtained.

Finally, regarding the scientific applications, we can see an average reduction of 23% in network traffic (see *AvgA* in Figure 9). More specifically, the applications Raytrace, Ocean and QSort present reductions of 23%, 1% and 45%, respectively. As before, there is a correlation between the fraction of the execution time devoted to lock synchronization and the amount of network traffic that is saved. For instance, Ocean presents the lowest reduction in network traffic since less than 5% of its execution time (see Figure 8) is spent on locks.

3) Energy Efficiency: Finally, we also consider the benefits in energy efficiency that our proposal entails. More specifically, we present in Figure 10 the normalized energy-delay^2 product (ED^{2}P) metric for the full CMP. To account for the energy consumed by the *GLocks* architecture (the *G-lines*-based network described in Section III-A), we extend the Sim-PowerCMP with the consumption model of *G-lines* and controllers employed in [21].

As in the previous two sections, all results in Figure 10 have been normalized with respect to the *MCS* case. As it can be observed, important improvements in the ED^{2}P metric of the whole CMP are achieved when applying

our proposal. In particular, the *GLocks* mechanism brings average improvements in ED^2P of 78% and 28% for the microbenchmarks and real applications, respectively. The SCTR, MCTR, DBLL, PRCO and ACTR microbenchmarks show reductions of 72%, 83%, 75%, 65% and 96%, respectively. Additionally, reductions of 50%, 10% and 25% are achieved for Raytrace, Ocean and QSort.

In general, the magnitude of these savings is directly related to the extents of the improvements in execution time and network traffic previously reported. We have found that when the *GLocks* mechanism is employed, the number of instructions executed per lock acquisition and release operation is drastically reduced. Note that while *MCS Locks* must deal with a distributed queue of waiting threads requesting the lock, *GLocks* only need two assignment instructions on two registers to notify the arrival to the lock and the subsequent release operation (see Section III-C). Obviously, less instructions executed means less energy consumed in the processor cores.

Moreover, since we reduce the latency of lock acquisitions, the busy-wait process is also shortened with *GLocks*. While busy-waiting, a processor core repeatedly access the L1 cache to check the value of a shared variable. In this way, shorter busy-waiting implies less accesses to the L1 cache, and therefore, less energy consumed in this structure. Finally, given the fact that our proposal skips the memory hierarchy, we save all the energy derived from coherence activity when locks are executed. In particular, we remove all of the L1 cache misses related to lock operations and the corresponding messages transferred across the interconnect. This brings reductions in the energy consumed at the L2 cache banks and the interconnection network.

V. Conclusions

Lock contention is recognized as a key constraint to performance and scalability on many-core CMPs when trying to exploit thread-level parallelism. In this paper we have proposed *GLocks*, a new hardware-supported implementation for highly-contended locks. *GLocks* deploys a dedicated on-chip network built with existing technology with minimal impact on energy consumption or die area. The use of a token-based messaging-protocol atop this network provides an extremely efficient and completely fair behavior. Furthermore, even though resources required to build this network grow with the number of supported locks, a deep analysis of some parallel applications discloses a reduced number of highly-contended locks in most cases. In this sense, our proposal is a hybrid approach which combines the devised *GLocks* mechanism with *Simple Locks* enhanced with the `test-and-test&set` optimization. While *GLocks* provide lightning-fast lock acquisition and release for highly-contended locks, the *Simple Locks* result in the best performance for low-contended locks. Performance results obtained on a simulated 32-core CMP with a 2D-mesh data network for a set of microbenchmarks and real applications corroborate our statements. An average reduction of 42% and 14% in execution time, an average reduction of 76% and 23% in network traffic, and an average reduction of 78% and 28% in the energy-delay^2 product (ED^2P) metric for the full CMP are achieved, for the microbenchmarks and the real applications, respectively.

As future work, we plan to complete this study by extending the applicability of *GLocks*. First, since the current *G-line*-based network design limits the maximum size of the CMP, it becomes necessary to somehow extend the mechanism to support larger CMPs. To do this, there are two possible paths to explore: a hierarchical *G-line*-based network and longer *G-line* latencies. Second, the current *GLocks* mechanism does not consider multiprogrammed workloads. To deal with them, a few *GLocks* could be statically or dynamically shared among all of the workloads.

Acknowledgments

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants "CSD2006-00046" and "TIN2009-14475-C04". José L. Abellán is supported by fellowship 12461/FPI/09 from Fundación Séneca - Agencia de Ciencia y Tecnología de la Región de Murcia (II PCTRM 2007-2010). Finally, we would like to thank Fabrizio Petrini for his invaluable contribution at the early stage of this work.

References

[1] P. Conway, "Blade Computing with the AMD Magny-Cours Processor," in *Proceedings of the 21st Symposium on High Performance Chips*, 2009.

[2] Single-chip Cloud Computer. [Online]. Available: http://techresearch.intel.com/articles/Tera-Scale/1826.htm

[3] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyakis, "Comparing Memory Systems for Chip Multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35(2), pp. 358–368, 2007.

[4] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[5] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing Lock Contention in Multithreaded Applications," in *Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[6] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of 22nd International Symposium on Computer Architecture*, 1995.

[7] R. Chang, N. Talwalkar, C. Yue, and S. Wong, "Near Speed-of-Light Signaling over On-Chip Electrical Interconnects," *IEEE Journal of Solid State Circuits*, vol. 38(5), pp. 834–838, 2003.

[8] A. Flores, J. L. Aragón, and M. E. Acacio, "Sim-PowerCMP: A Detailed Simulator for Energy Consumption Analysis in Future Embedded CMP Architectures," in *Proceedings of 21^{st} International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[9] T. E. Anderson, "The Performance Implications of Spin-Waiting Alternatives for Shared Memory Multiprocessors," in *Proceedings of the Intel Conference on Parallel Processing*, 1989.

[10] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems*, vol. 9(1), pp. 21–65, 1991.

[11] A. Kägi and D. Burger and J. R. Goodman, "Efficient Synchronization: Let Them Eat QOLB," in *Proceedings of the 24^{th} International on Computer Architecture*, 1997.

[12] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal, "Smartlocks: Self-Aware Synchronization through Lock Acquisition Scheduling," in *Proceedings of the 7^{th} IEEE/ACM International Conference on Autonomic Computing and Communications*, 2010.

[13] B.-H. Lim and A. Agarwal, "Reactive Synchronization Algorithms for Multiprocessors," *ACM SIGPLAN Notices*, vol. 29(11), pp. 25–35, 1994.

[14] C.-C. Kuo, J. Carter, and R. Kuramkote, "MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing," in *Proceedings of 5^{th} International Symposium on High Performance Computer Architecture*, 1999.

[15] C. Wagner and F. Mueller, "Token-based Read/Write-Locks for Distributed Mutual Exclusion," in *Proceedings of 6^{th} International Euro-Par Conference on Parallel Processing*, 2000.

[16] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "An Efficient Synchronization Technique for Multiprocessor Systems on-Chip," *ACM SIGARCH Computer Architecture News*, vol. 34(1), pp. 33–40, 2006.

[17] A. Jose and K. L. Shepard, "Distributed Loss-Compensation Techniques for Energy-Efficient Low-Latency On-Chip Communications," *IEEE Journal of Solid State Circuits*, vol. 42(6), pp. 1415–1424, 2007.

[18] H. Ito, M. Kimura, K. Miyashita, T. Ishii, K. Okada, and K. Masu, "A Bidirectional-and Multi-Drop-Transmission-Line Interconnect for Multipoint-to-Multipoint On-Chip Communications," *IEEE Journal of Solid State Circuits*, vol. 43(4), pp. 1020–1029, 2008.

[19] R. Ho, T. Ono, F. Liu, R. Hopkins, A. Chow, J. Schauer, and R. Drost, "High-Speed and Low-Energy Capacitively-Driven On-Chip Wires," in *Proceedings of the IEEE Solid-State Circuits Conference*, 2007.

[20] E. Mensink, D. Schinkel, E. Klumperink, E. Tuijl, and B. Nauta, "A 0.28pf/b 2gb/s/ch Transceiver in 90nm Cmos for 10mm On-Chip Interconnects," in *Proceedings of the IEEE Solid-State Circuits Conference*, 2007.

[21] T. Krishna, A. Kumar, P. Chiang, M. Erez, and L.-S. Peh, "NoC with Near-Ideal Express Virtual Channels using Global-Line Communication," in *Proceedings of 16^{th} IEEE Symposium on High Performance Interconnects*, 2008.

[22] J. L. Abellán, J. Fernández, and M. E. Acacio, "A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs," in *Proceedings of the 39^{th} International Conference on Parallel Processing*, 2010.

[23] C. J. Huges, V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, vol. 35(2), pp. 40–49, 2002.

[24] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, "Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27(1), pp. 23–34, 1999.

[25] R. Rajwar and J. Goodman, "Transactional Lock-free Execution of Lock-based Programs," in *Proceedings of 10^{th} Annual Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.