EVALUATION OF ARCHITECTURAL SUPPORTS FOR FINE-GRAINED SYNCHRONIZATION MECHANISMS

Tiziano De Matteis, Fabio Luporini, Gabriele Mencagli and Marco Vanneschi
Department or Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
Email: {dematteis, luporini, mencagli, vannesch}@di.unipi.it

ABSTRACT
The advent of multi-/many-core architectures demands efficient run-time supports to sustain parallel applications scalability. Synchronization mechanisms should be optimized in order to account for different scenarios, such as the interaction between threads executed on different cores as well as intra-core synchronization, i.e. involving threads executed on hardware contexts of the same core. In this perspective, we describe the design issues of two notable mechanisms for shared-memory parallel computations. We point out how specific architectural supports, like hardware cache coherence and core-to-core interconnection networks, make it possible to design optimized implementations of such mechanisms. In this paper we discuss experimental results on three representative architectures: a flagship Intel multi-core and two interesting network processors. The final result helps to untangle the complex implementation space of synchronization mechanisms.

KEY WORDS
Synchronization, Locking, Simultaneous Multi-Threading, Busy-Waiting, Multi-cores, Network Processors.

1 Introduction

Recent technological advancements led to a number of architectures that can be used to accelerate compute-intensive applications. Parallel, possibly fine-grained applications require specialized mechanisms to scale with parallelism degrees that became significant even inside a single on-chip platform. In this context, the problem of efficient thread cooperation is a key factor for application performance. Run-time supports for parallel computations should provide a set of highly optimized basic mechanisms such as locks, notification primitives, barriers, semaphores as well as lock-free data structures.

In current shared-memory architectures, the plethora of different configurations in terms of cache hierarchies, presence of multiple thread contexts per core, interconnection networks and other architectural facilities, renders the problem of finding the best implementation of basic mechanisms extremely complex. Moreover, the best implementation might not exist, but alternative techniques should be compared in a multidimensional space of metrics in which the best trade-off solutions need to be clearly identified.

In order to explore alternative implementations, we study two synchronization mechanisms. Consider two threads that are running their own computation, but at a certain point they need to synchronize in order to proceed with the execution. In the first scenario, a thread waits until a particular condition is satisfied. The role of the peer thread is to inform it as soon as the condition becomes verified. We call this general type of interaction *one-way notification*. In the second scenario, a pool of threads access a critical section of code. In this case a *lock* is necessary to preserve the computation consistency. In this paper we do not cover all the synchronization mechanisms available for building parallel applications, but we highlight the importance of optimizing a common critical phase, i.e. *the busy-waiting of threads inside synchronization primitives*.

To have a sufficiently large space of optimizations, we study three test-bed architectures: a "classic" *Intel Xeon* multi-core and two novel network processors, the *Tilera Tile64Pro* and the *Broadcom XLP 432*. We present different versions of the mechanisms: using classic spin-loops on shared flags, advanced assembler instructions to be notified of a cache line modification, and core-to-core interconnection networks to exchange small messages directly at the firmware level. As we will see, plain spin-wait loops may be unsatisfying under certain circumstances, and the presence of the *Simultaneous Multi-Threading* (SMT) technology exacerbates this problem. Other implementations, though less effective from the responsiveness viewpoint, are less resource consuming and, thus, able to provide a "graceful" way to synchronize threads on different contexts of the same core. In this paper we describe several experiments and we try to provide a guideline to simplify the jumble of alternative implementations by highlighting the strengths and the weaknesses of different solutions.

The paper is organized as follows: Section 2 presents the synchronization mechanisms we are studying; Section 3 provides a description of the platforms we used in our experiments; Section 4 shows the implementation details and the experimental results. Finally, Section 5 summarizes the main findings and concludes the paper.

2 Synchronization mechanisms

Especially for highly parallel shared-memory architectures, lightweight synchronization primitives are fundamental to achieve scalable implementations of fine-grained parallel programs. Therefore, these mechanisms require a clever design in order to minimize the architectural overhead. To approach this problem, we explore the space of optimizations and architectural supports for two mechanisms:

- a ***one-way notification***, that enables threads to wait until a particular event occurs. For instance, a producer thread can notify a worker that a new task has to be computed. Such mechanism can support operations to notify either a single or all threads that are waiting on the same condition;

- a ***locking mechanism***, that serializes the execution of a critical section of code. Alternative implementations can be compared against the impact on the memory and caching sub-system, the number of invalidation messages, and the fairness of the lock acquisition.

In the next section we describe the benchmark applications that we have used to compare different implementations of the aforementioned mechanisms.

2.1 Benchmark applications

We present two synthetic benchmark applications. The former application has been designed to assess one-way notification; a similar test is also used in [1] for analogous purposes. The latter one targets locking mechanisms, and its design has been inspired by [5]. To compare different implementations, we define a set of meaningful metrics and we collect the corresponding measurements.

First benchmark. A one-way notification mechanism (generally named *sync_var* in this paper) allows the involved threads to perform the following set of operations:

- a *wait* primitive is used to make the calling thread waiting for a given event. Since the scope of this paper is to discuss fine-grained synchronization techniques, we are interested in lightweight implementations based on busy-waiting, i.e. the waiting thread is not suspended, but it continuously or periodically checks the wait condition;

- a *notify* primitive is used to signal one of the waiting thread that it can continue the execution. It is also possible to wake up all threads waiting for the same condition, by executing a collective *notifyAll*.

The first benchmark consists in a notifier thread that executes the following sequence of actions: (i) a computation that involves a set of arithmetic operations over the elements of an array; (ii) a notify call to wake up a thread waiting on the sync_var. The waiting thread performs a wait primitive on the sync_var. This test is repeated for a fixed number of iterations N. For this benchmark we define the following metrics:

- the *mean wakeup time* T_{wake}, i.e. the average time to advise a thread that an event occurred, starting from the instant in which the notify primitive is executed by a peer thread. This metric is extremely important to assess the responsiveness of the mechanism;

- the *mean call time* T_{call}, i.e. the average time to execute the notify primitive by the calling thread;

- the *mean completion time* T_C, i.e. the total elapsed time to complete the benchmark;

As an alternative yet similar experiment, we analyze a second scenario wherein a group of n_w threads is waiting for an event and is collectively notified by a notifier thread. In this case, we introduce another important metric:

- the *mean wakeup-all time* $T_{wake\text{-}all}(n_w)$, i.e. the average time to advise all n_w threads that an event occurred, starting from the instant in which the $notifyAll$ primitive is executed by a peer thread.

The crucial part of this mechanism is the busy-waiting phase. The responsiveness of a specific implementation can be evaluated according to the T_{wake} metric. Implementations with lower T_{wake} (or $T_{wake\text{-}all}(n_w)$) are usually preferred: after the notification, the awaken thread needs to resume the execution as soon as possible. On the other hand, there are also subtle situations in which faster implementations are not always the most effective solution. Think about two threads executed on the same core (i.e. mapped onto distinct physical contexts, as in a SMT-enabled architecture). Besides considering T_{wake}, it is important to evaluate how much the waiting thread consumes the core resources during the busy-waiting phase (e.g. pipe stages, instructions and operands queues, and register renaming logic), degrading the performance of the peer thread in execution on the same core. In this case, the completion time T_C becomes a key metric to consider as well.

Second benchmark. The second benchmark aims to compare different implementations of locking. In the literature there is a large number of studies comparing several locking implementations targeting large-scale multiprocessors (e.g. [8, 2, 9, 7]). In this paper we investigate the potential of advanced hardware facilities to implement novel, lightweight locking mechanisms.

The benchmark consists in a pool of P threads that attempt to acquire a lock, do a fixed amount of work simulating a critical section of length C, and release the lock. The number of lock acquisitions N is maintained as the number of threads increases. We are interested in the execution time of the experiment and in the lock overhead, captured by the following metric:

- the *mean contention time* T_{lock}, i.e. a measure of the contention cost, defined as follows:

$$T_{lock} = \frac{\text{Execution\_Time}}{N} - C$$

Also for the locking mechanism the busy-waiting phase is extremely critical: as demonstrated in [8, 7], aggressive techniques generate a huge number of cache invalidation messages. Other techniques try to reach a desired trade-off between the mechanism responsiveness and the induced network traffic on the interconnection structure.

3 Test-bed Architectures

Our experiments will be executed on three distinct on-chip architectures.

Intel Xeon. Our Intel platform is composed of two Xeon E5-2650 (the details of a single CPU are depicted in Figure 1). Each CPU exploits the Sandy Bridge technology and consists in a multi-core architecture featuring 8 cores operating at 2 Ghz clock rate. Each core enables the execution of up to 2 SMT contexts (Hyper-Threading) and has access to a private L1 and L2 cache of size 32 KB and 256 KB. A L3 cache of size 20 MB is shared among the eight cores through a scalable ring interconnect. The two CPUs use the QuickPath structure to communicate and implement a unique shared-memory architecture.

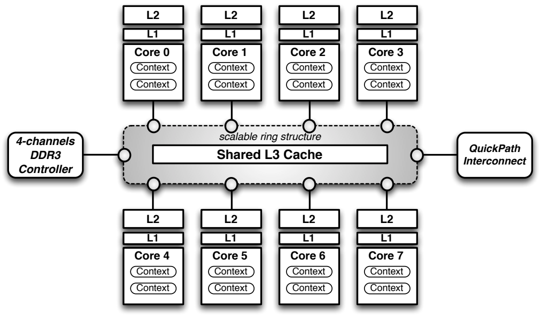


Figure 1: Overview of an Intel Xeon E5-2650.

In this architecture the analysis of a mechanism is obviously affected by the actual scope of the synchronization, that may occur among two threads of the same core (*SMT-case*), on different cores of the same CPU (*intra-CPU*), or even on different CPUs (*inter-CPU*).

Tilera Tile64Pro. Tile64Pro is a NUMA network processor consisting of 64 cores (also called *tiles*) having access to a shared main memory among four distinct memory controllers as shown in Figure 2. Each core is a 32-bit VLIW processor operating at 866 Mhz clock rate. It provides a single-thread context without floating-point unit, a private L1 and L2 cache of size 16 KB and 64 KB, and a switch unit to interface the on-chip network.

The architecture implements a directory-based cache-coherence protocol. Each cache line may be allocated in several caches, but it is associated with a single *home core* which maintains its original copy and the directory information. This technique aims at implementing a *logical distributed L3 cache* among the 64 cores of the architecture, i.e. miss-after-read cache accesses are handled by asking the home core for the desired cache line, rather than by issuing requests directly to the main memory.

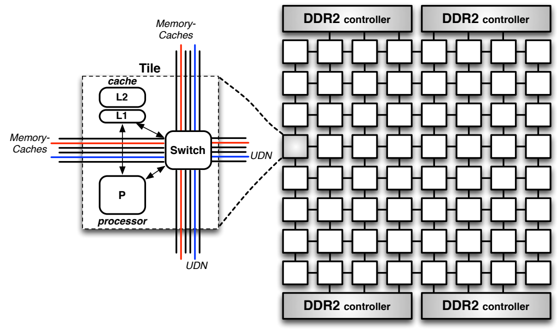


Figure 2: Tile64Pro Network Processor.

One of the most interesting features of this architecture is the degree of freedom left to the programmer in terms of caching strategies. When a data structure is dynamically allocated, the programmer can specify different caching techniques, e.g.: (i) specify the home core to host the directory information of all the cache lines of that data structure; (ii) disable the local caching (all read/write requests are issued directly to the home core); (iii) completely disable the cache (accesses become single-word main memory accesses); (iv) disable the automatic coherence (i.e. for incoherent memory areas, the coherence should be manually provided by the programmer through flush and invalidation primitives).

The Tile64Pro uses some mesh networks to interconnect the various node components, each one carrying out a specific purpose. One of this networks, called *User Dynamic Network* (**UDN**), may be exploited at the user-level to perform a fast exchange of small messages (up to 20 words) between cores, through send and receive primitives on hardware-level queues associated to the cores.

In Section 4 we will see how this architectural feature paves the way to interesting implementations of the synchronization mechanisms.

Broadcom XLP. The Broadcom XLP 432 is a SMP network processor consisting of 8 identical cores. Figure 3 illustrates the main components of the architecture. Each core is a MIPS 64-bit out-of-order processor operating at 1 Ghz clock rate and featuring a dedicated floating-point unit. Each processor enables up to four SMT contexts (called *NxCPU*) and has access to a private L1 and L2 cache of size 32 KB and 512 KB. Moreover, a 8 MB L3 cache is shared among the 8 cores. A small set of co-processors (called engines) is provided to accelerate typical networking functionalities (e.g. regular expressions, compression, security and packet inspection).

A remarkable feature of this architecture is the *Fast Messaging Network* (**FMN**), an interconnection subsystem that enables a unified approach to the communi-

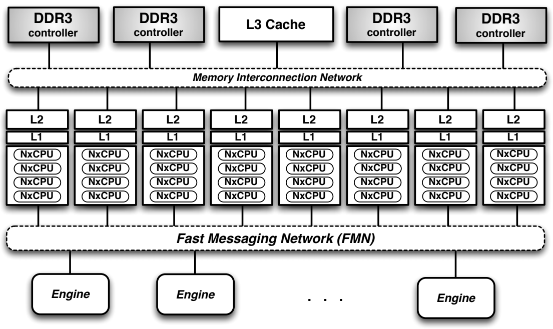


Figure 3: Broadcom XLP 432 Network Processor.

cation between NxCPUs and the co-processors. From an abstract viewpoint, the network resembles the Tilera UDN: it provides an alternative user-level communication mechanism (instead of using shared variables). The rationale proposed in the Broadcom architecture (and similarly on other network processors like Tilera) is that under certain conditions the synchronization efficiency can be improved by a methodical use of the available on-chip interconnection networks.

The Broadcom XLP architecture is an interesting case. In fact, albeit it owns the typical features of other network processors (e.g. the presence of co-processors and dedicated interconnection sub-systems), the Broadcom chip shares peculiar aspects of traditional off-the-shelf multi-/many-core components, such as a complex memory hierarchy composed of several cache levels (dedicated or shared among the architectural cores).

4 Experiments

In this section we discuss the implementations and the experiments on the three architectures described in Section 3. We highlight that we are not looking for a mere comparison between the three architectures, which is certainly unfair due to their completely different nature. However, we claim that such experiments are important to understand the rationale behind the different technological choices and the physical facilities provided on such platforms.

4.1 One-way notification

The general implementation of a synch_var consists in a data structure composed of the following fields:

- a circular buffer of identifiers that represent the currently waiting threads. Since we are interested in the design of run-time supports for parallel programs, the cores of a parallel architecture are not multiprogrammed, i.e. a parallel computation is executed according to a parallelism degree that implies the allocation of n threads on n different cores (or physical contexts on a SMT-enabled architecture). Therefore, each thread identifier corresponds to a specific physical core/context on the underlying architecture;
- a set of additional variables that indicate: the head and the tail fields of the circular buffer, and the actual and the maximum number of waiting threads.

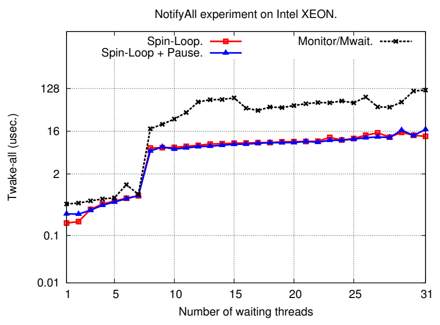
The notify, notifyAll and the wait primitives cause the modification of the data structure. A synch_var is always used inside critical sections protected by a lock.

Intel Xeon. A "careful" busy-waiting implementation is extremely important for this mechanism. We tested three different techniques for this architecture. The first two rely on a classic spin-loop mechanism. The circular buffer is composed of cache-aligned boolean flags, each of them associated to one waiting thread. In the wait primitive, the calling thread chooses one of the free flags, sets its value to *false* and cyclically tests it until it becomes *true*. The notify primitive sets to *true* the flag associated to the first waiting thread, in order to wake it up. The second implementation is obtained as a slight modification of the first one, by inserting the `pause` instruction in the waiting loop. This is an assembler instruction [6] that introduces a light delay in the loop and de-pipelines its execution, preventing it from aggressively consume processor resources.

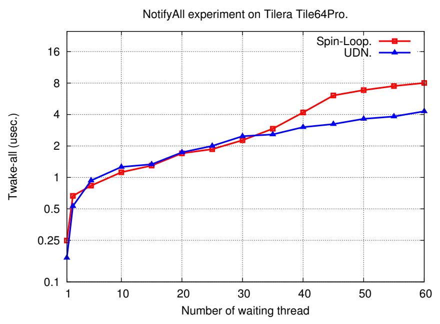
The third version uses the *monitor* and *mwait* assembler instructions, which have been already investigated in [1]. These instructions can be used as an alternative way to perform a busy-waiting on the flag associated to the waiting thread. The `monitor` instruction takes a memory address and supervises the referenced location for the occurrence of a write activity. The `mwait` instruction places the calling processor in a "performance-optimized" state until either a write to the region supervised by `monitor` occurs or a generic interrupt reaches the processor [6]. On a hyper-threaded processor the `mwait` causes a thread to relinquish all core resources that are shared with the other contexts. In addition, it is worth to notice that since `mwait` may return even in presence of a hardware interrupt, the `monitor/mwait` instructions need to be executed in a loop to ensure the content of the monitored location actually changed. Besides the basic semantics of these two instructions, the Intel's documentation [6] does not give sufficient details about the effects and cost of executing `monitor/mwait` instructions. However, when it is possible we will try to conjecture the reasons behind the collected experimental results.

At the present moment `monitor/mwait` instructions are restricted to be executed solely in kernel space. Thereby, in order to implement a sync_var mechanism based on these two instructions, there are two possible solutions:

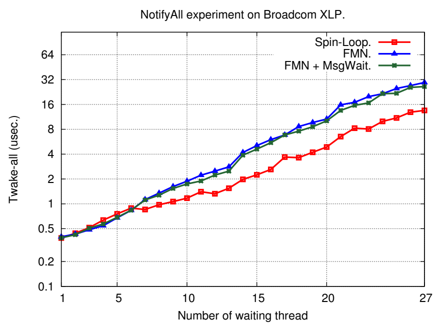
- extend the Linux Kernel with a pair of system calls that wrap the privileged instructions;
- execute the whole program in kernel space.



(a) Intel Xeon.



(b) Tilera Tile64Pro.



(c) Broadcom XLP.

Figure 4: NotifyAll experiments on the three test-bed architectures.

Unlike the authors of [1], we chose the second option because we feel `monitor/mwait` instructions will be released in user-space in the next future. To accomplish this, we used a patched version of the Linux Kernel that allows us to use these instructions without specific system calls (see *Kernel Mode Linux* [10]).

Tables 1, 2 and 3 show the results of the first benchmark with one notifier thread and one waiting thread (τ denotes the clock cycle).

Version	**Intra-Core**	**Inter-Core**	**Inter-CPU**
`Spin Loop`	169 τ	460 τ	4197 τ
`Spin+Pause`	174 τ	605 τ	3255 τ
`Monitor/Mwait`	1052 τ	1069 τ	2339 τ

Table 1: Notify experiments on Intel: T_{wake}.

The spin-loop implementation provides the best T_{wake} when the two threads are allocated on the same core or the same CPU, whereas the variant with the `pause` instruction makes slightly worse the wakeup time. This is an expected outcome, because spin-loop allows threads to react immediately to the event triggered by a *notify* primitive. The wakeup latency is smaller in the *intra-core* case because the notification can be completed within the L1 cache, while in the *inter-core* case the flag modification passes through the nearest shared level of cache (L3) to be visible to the notified thread. The `monitor/mwait` have to pay an extra-cost for re-obtaining the computational resources dynamically relinquished while in the performance-optimized state (e.g. instruction schedulers and reservation stations). In the intra-core case the wakeup latency is one order of magnitude greater than using spin-loops (in the inter-core case we have a 43% increase). However, when threads are allocated on different CPUs the `monitor/mwait` give strikingly better results. This is probably due to the not clear behavior of the QuickPath interconnection, though further investigations are still needed to elucidate the precise nature of this result.

The completion time results demonstrate that the spin-loop implementation is the most aggressive in term of resource consumption. In fact, in the intra-core case it exhibits the worst completion time: this is due to the fact that in modern out-of-order processors a spin loop is unrolled multiple times (since no data dependencies are found) and the branch is easily predicted. Hence, the spinning thread, though not performing useful work, fills the pipeline with a significant number of instructions. These end up interfering with other threads in execution on the same core, causing an overall slow down. The use of the `pause` instruction mitigates the problem, but does not solve it completely. The best solution is represented by the implementation that utilizes the `monitor/mwait` instructions.

Version	**Intra-Core**	**Inter-Core**	**Inter-CPU**
`Spin Loop`	2359 ms	1703 ms	1715 ms
`Spin+Pause`	1872 ms	1708 ms	1712 ms
`Monitor/Mwait`	1704 ms	1711 ms	1717 ms

Table 2: Notify experiments on Intel: completion time.

Version	**Intra-Core**	**Inter-Core**	**Inter-CPU**
`Spin Loop`	42 τ	69 τ	472 τ
`Spin+Pause`	36 τ	63 τ	437 τ
`Monitor/Mwait`	46 τ	62 τ	98 τ

Table 3: Notify experiments on Intel: T_{call}.

In terms of mean call time, the three solutions are comparable in the intra- and inter-core cases, while the `monitor/mwait` implementation performs surprisingly better in the *inter-CPU* case. As previously mentioned, the scarce knowledge about the QuickPath network does not allow us to further explain this fact.

The same benchmark has been repeated with a parametric set of waiting threads collectively notified by a single notifyAll call. Up to 15 waiting threads the affinity is set such that only one context is used for all the cores of the architecture. With more than 15 waiting threads, we use the second context starting from the first core (the one in which

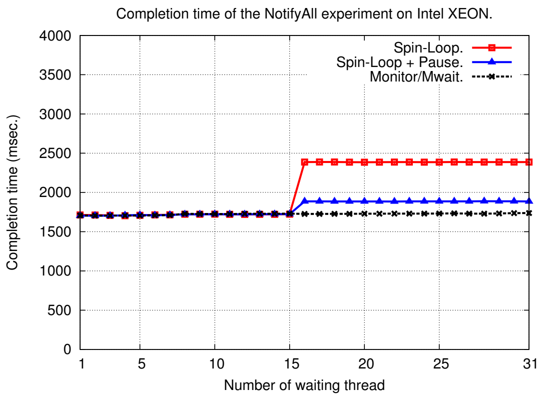


Figure 5: Completion time T_C of the NotifyAll experiment on Intel.

the notifier thread is executed). The results are shown in Figure 4a. Although the `monitor/mwait` implementation is the best one in terms of T_{wake} between two threads on different CPUs (Table 1), the notifyAll implementation based on these two instructions provides a greater $T_{wake\text{-}all}$ compared to the spin-loop alternatives. Regarding the completion time $T_C(n_w)$, the results exposed in Figure 5 reflect the ones of the first experiment. For spin-loop based mechanisms the $T_C(n_w)$ shows a significant, fixed increment for $n_w > 15$, i.e. when one waiting thread is allocated on the same physical core of the notifier thread. In the same condition, the `monitor/mwait` implementation gives a unique and not questionable advantage, being capable of maintaining a constant completion time compared to the other solutions.

Tile64Pro. We compare two implementations of the synch_var on the Tile64Pro. The first one implements the busy-waiting through a classic spin-loop, as the one described in the previous section for the Intel Xeon. The second one proposes the exploitation of the UDN network.

Instead of using shared flags, in the UDN implementation we use the explicit communication between cores. The synch_var consists in a circular buffer of *Dynamic-Headers*. A DynamicHeader is a data structure of 4 bytes that represents the coordinates of a core on the mesh network. During the wait primitive, the calling thread inserts a DynamicHeader representing its core on the circular buffer, and waits for the reception of a special message by executing the `udn_receive()` call. The calling thread is still in execution, but it waits for the reception of a message from one of its four hardware-level input queues (the queue identifier is passed as an argument to the wait). During a notify operation, the notifier extracts the DynamicHeader of the first waiting thread and executes the `udn_send()` call. The procedure transmits a single-word message to one of the four message queue of the destination core.

Table 4 and Table 5 show the experimental results. On the Tilera architecture the mean wakeup time is influenced by the distance between the notifier and the awaken

Allocation Policy	$\mathbf{T_{wake}}$ (cycles)	$\mathbf{T_{call}}$ (cycles)
`DEFAULT`	216 τ	171 τ
`NOTIFIER`	171 τ	132 τ
`NOLOCAL_CACHE`	324 τ	311 τ
`NO_COHERENCE`	495 τ	559 τ
`NO_CACHE`	605 τ	599 τ

Table 4: Notify experiments on Tile64Pro: Spin-Loop implementation.

thread on the mesh network. Therefore we execute the experiment several times considering different allocation of the two threads and providing the average measurements of the wakeup time and the call time. Moreover, in order to exploit the flexibility of the Tilera architecture, we study the mechanism by considering the following caching policies of the synch_var data structure:

- `DEFAULT`: the home tiles of the synch_var cache lines are distributed according to a default hashing strategy;
- `NOTIFIER`: the home tile of the synch_var cache lines is the core that hosts the notifier thread (we assume that the notifier is fixed throughout the execution);
- `NOLOCAL_CACHE`: all the accesses to any synch_var cache line are routed to the corresponding home tile, chosen according to the `DEFAULT` policy without taking an own copy on the local L1 and L2 caches;
- `NO_COHERENCE`: caching of the synch_var is enabled without the automatic maintenance of the coherence. Coherence is explicitly provided by the programmer through the execution of flushing and invalidation operations;
- `NO_CACHE`: the caching of the synch_var is completely disabled. All the accesses to the data structure are routed to the main memory.

The best results (summarized in Table 4 and 5) in terms of T_{wake} and T_{call} are obtained with the `NOTIFIER` policy. In this way all memory accesses of a notify call are executed by the notifier thread on its local caches, without any remote access to update the home copy and the directory information. This is not the case when the home tile is a third party between the notifier and the currently awaken thread (as in the `DEFAULT` case). All other policies correspond to a slower implementation of the mechanism. The spin-loop implementation with the `NO_CACHE` policy implies a T_{wake} and a T_{call} four times higher than with the `NOTIFIER` policy.

From Table 4 and 5 emerges that the UDN implementation is in general faster than the classic spin-loop. The UDN network provides an efficient way to exchange small messages between cores without resorting on any level of shared memory. Moreover, instead of continuously looping

Allocation Policy	$\mathbf{T_{wake}}$ (cycles)	$\mathbf{T_{call}}$ (cycles)
DEFAULT	148 τ	120 τ
NOTIFIER	98 τ	88 τ
NOLOCAL_CACHE	185 τ	194 τ
NO_COHERENCE	190 τ	267 τ
NO_CACHE	283 τ	361 τ

Table 5: Notify experiments on Tile64Pro: UDN implementation.

on a shared flag, the `udn_receive()` is a graceful way to wait for an event, without producing additional remote accesses during the waiting phase. This is the reason because the `NOLOCAL_CACHE`, `NO_COHERENCE` and `NO_CACHE` policies are in general faster in the UDN implementation w.r.t the spin-loop case.

In the second experiment we investigate the scalability of the mechanism: at each iteration the notifier thread performs a notifyAll instead of a single notify. We limit the maximum number of waiting threads to 60, since in the actual configuration three cores are outside our data-plane and they are dedicated to operating system processes and network management activities. The results are shown in Figure 4b. As we can see the two implementations provide very similar results up to 30 waiting threads. For a greater number of threads the UDN version outperforms the spin-loop implementation. In fact, besides the cache line of the head and tail fields, the spin-loop version requires to modify a large set of flags allocated on distinct cache lines, causing a non-negligible coherence traffic on the mesh to update the single copies of the flags. With the UDN approach the notifyAll can be implemented definitively better, exploiting on a hardware facility that provides an efficient way to exchange single-word messages instead of whole cache lines as in the spin-loop case.

Broadcom XLP. Similarly to the Tilera architecture, on the Broadcom XLP we compare the spin-loop implementation and a version that exploits the FMN interconnection structure between NxCPUs.

The FMN implementation is very similar to the UDN version on the Tile64Pro. A circular buffer of message headers is provided in order to identify the NxCPUs of the currently waiting threads. During a wait operation, the calling thread inserts the header of its NxCPU on the circular buffer. The identifier of the first thread to be awaken is extracted during a notify call.

The FMN network accepts two kinds of messages from an agent (i.e. a NxCPU or an engine): push and pop requests. An agent can send a message by sending a push request to a push queue coupled with the specific destination agent. Pop requests are a way to implement a decoupled communication: an agent can issue a push request to send a message to a pop queue. Pop requests are used to retrieve messages from a pop queue. In order to implement the synch_var mechanism, we exploit send and receive operations over push queues. From a programming interface viewpoint, the FMN primitives provide a non-blocking way to interact with the interconnection structure. `xlp_message_send()` and `xlp_message_receive()` are non-blocking operations: e.g. a receive returns 0 if a message has been read or an error value if there is no message in the selected queue.

In the FMN case we consider two implementations. In the first one, the notifier thread performs a `xlp_message_send()` using the first header extracted from the synch_var. The wait primitive consists in the insertion of the header in the circular buffer and the execution of the `xlp_message_receive()`. Due to the non-blocking semantics, the busy-waiting is implemented by using the receive operation inside a while loop, in which we test the result of the receive (the loop ends when a message has been received). This waiting phase can be optimized as described in the following pseudo-code:

```
while(xlp_message_receive(...)==-1){
  xlp_message_wait(...);
}
```

Inside the loop phase we can execute the `xlp_message_wait()` call. The semantics of this call is to force the calling thread to release all the processor resources until an event occurs, such that the presence of a new message or the occurrence of a hardware interrupt. We compare different FMN implementations of the synch_var: the first one does not use the `xlp_message_wait()`, the second one (denoted by `FMN+MsgWait`) exploits this optimization.

Version	Intra-Core	Inter-Core
Spin Loop	168 τ	386 τ
FMN	295 τ	399 τ
FMN+MsgWait	269 τ	387 τ

Table 6: Notify experiments on Broadcom XLP: T_{wake}.

The comparison between the spin-loop and the FMN implementations is described in Table 6 for the T_{wake} and in Table 7 for the T_{call}. We consider the *intra-core* case, in which the notifier and the awaken threads are executed on different thread contexts of the same core, and the *inter-core* case in which the two threads are executed on different cores. As we can observe, while for the *inter-core* case the three implementations behave similarly, the spin-loop version provides the best T_{wake} and T_{call} results for synchronizing two threads on the same core. In fact with the spin-loop technique the shared flag is accessed by the two threads on the same L1 cache, with a very small latency than accessing a remote push queue on the FMN network.

Figure 4c depicts the notifyAll results. For a similar reason to the Tilera experiments, we limit the number of waiting threads to 27, since one core of the architecture is

Version	Intra-Core	Inter-Core
`Spin Loop`	100 τ	195 τ
`FMN`	193 τ	304 τ
`FMN+MsgWait`	190 τ	309 τ

Table 7: Notify experiments on Broadcom XLP: T_{call}.

dedicated to operating system processes and interrupt handling. Up to 7 waiting threads the three implementations provide very similar results in terms of $T_{wake\text{-}all}$. By increasing the number of threads, the spin-loop implementation is still the best one (the `FMN+MsgWait` optimization achieves slightly better results compared to the basic `FMN` version). These results are diametrically different than the Tile64Pro experiments. The two network processors are extremely different, both in terms of core technology (several SMT contexts on the Broadcom XLP and no SMT support on the Tile64Pro) and memory hierarchy (large shared levels of cache on Broadcom XLP whereas Tilera features very small local caches). Moreover, the on-chip network topology is completely different, i.e. a mesh network on the UDN case and a ring-based topology for the FMN. This is a general reason for the different behavior of our implementations: shared flags are by far the most responsive way on the Broadcom XLP, especially when we need to synchronize a large number of threads. This means that "under load" the response time of the Memory Interconnection Network (Figure 3) is better than the message latency provided by the FMN network.

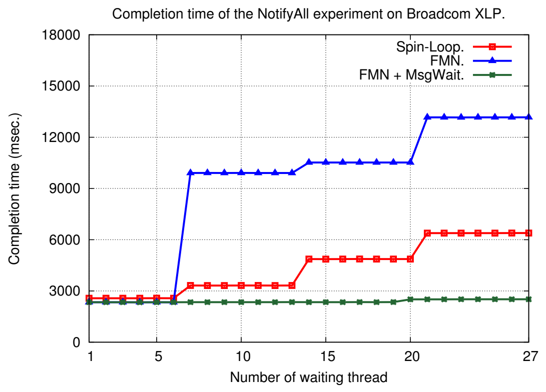


Figure 6: Completion time T_C of the NotifyAll experiment on Broadcom XLP.

As usual, we must be care that multiple thread contexts of the same core share processor resources. A highly responsive busy-waiting technique might be ineffective if the waiting thread makes slower the execution of other threads by consuming processor resources unnecessarily.

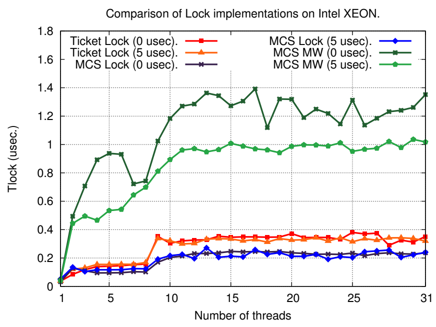
Let us consider the results shown in Figure 6. We measure the completion time of the notifyAll experiment. The notifier thread is executed on the first context of the first core. The waiting threads are mapped onto the cores in a round-robin fashion. As we can observe the completion time abruptly increases in correspondence to 7, 14 and 21 waiting threads. This is due to the mapping strategy onto the SMT contexts. From 7 to 13 threads, one waiting thread is executed on the same core of the notifier. From 14 to 20 threads, two waiting threads are executed on two contexts of the same core of the notifier. Finally, from 21 to 27 waiting threads, all the contexts of the first core are completely used. As we can notice, the best results in terms of completion time are achieved by using the `FMN+MsgWait` version. The completion time does not increase independently from the number of threads executed on the same core of the notifier thread. This demonstrates that a waiting phase implemented through a loop of `xlp_message_receive()` operations and the `xlp_message_wait()` optimization represents a good trade-off between: (i) mechanism responsiveness (however lower than the spin-loop case), and (ii) "lightness" of the waiting technique (spin-loop is a too aggressive and not suitable to synchronize threads within the same core).

4.2 Locking mechanisms

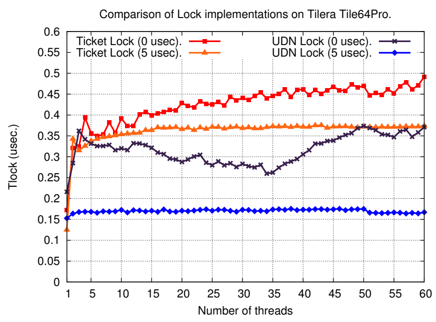
The experiments of a generic one-way notification mechanism gave us important insights about efficient ways to perform the wake up of a waiting thread. Careful busy-waiting techniques are also important for the design of locking mechanisms. Naive implementations tend to produce a large memory and interconnect traffic [8], introducing bottlenecks that become more pronounced with the number of threads that contend for the lock acquisition. In this section we apply the considerations emerged from Section 4.1 in order to design a locking mechanism such that:

- it provides a reasonable latency to acquire the lock in absence of contention;
- it is fair, i.e. each thread succeeds in acquiring the lock in a finite time interval;
- it scales with the number of threads that simultaneously concur to acquire the lock;
- the used busy-waiting technique represents an acceptable trade-off between *responsiveness* and *resource consumption*, i.e. it can be used to protect a critical section from the simultaneous access of multiple threads executed on different cores or executed on distinct thread contexts of the same core.

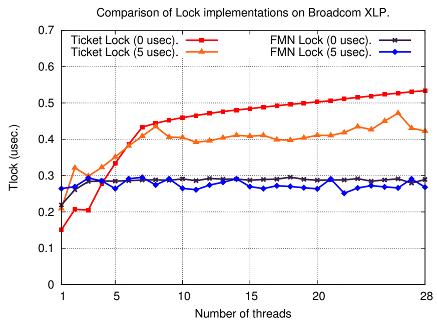
The mechanism that we are introducing is an extension of the *Mellor-Crummey-Scott* lock (shortly **MCS**) [8]. MCS lock is a well-known scalable approach based on a lock-free queue that ensures the FIFO ordering of the lock reception. It requires two atomic instructions to preserve the correctness of the queue access: a *swap* operation that atomically swaps the content of two memory locations, and a *compare-and-swap* instruction that atomically performs a comparison and a swap instruction.



(a) Intel Xeon.



(b) Tilera Tile64Pro.



(c) Broadcom XLP.

Figure 7: Locking experiments on the three test-bed architectures.

In the MCS algorithm the lock and unlock operations require an additional structure (called *record*) to be passed in addition to the queue address. The record contains a locally-accessible flag and a pointer to the next record. To acquire the lock, the thread performs the atomic swap between the tail field and a pointer to its record. If the old tail was NULL, the queue was empty and the thread acquires the lock. Otherwise the calling thread links its record to the next one and performs a busy-waiting on its local flag. To release the lock, the holding thread resets the flag of its successor. If no successor exists, the tail variable is set to NULL with the compare-and-swap instruction.

In the traditional version of the MCS lock, the busy-waiting phase is performed by spinning on a local flag. However the waiting phase can be implemented in different ways, by resorting on the techniques studied in Section 4.1. We describe the results of the locking benchmark presented in Section 2.1 on the three test-bed architectures. In the benchmark we consider critical sections of different lengths, i.e. 0 and 5 μsec, in order to evaluate the lock behavior with different contention levels. In addition, we compare the overhead and the scalability of different MCS implementations with the classic fair technique called *Ticket Lock* [2], in which two globally-accessible counters are properly used to serialize the access to the critical section.

Intel Xeon. For the Intel Xeon, an additional implementation of the MCS lock can be obtained performing the busy-waiting on the shared flag by means of the `monitor/mwait` instructions. We have compared the traditional ticket lock and the two versions of the MCS lock; the results are collected in Figure 7a. Apart from an increase on T_{lock} obtained when we start to allocate thread on different CPUs (when we use more than 8 threads), the ticket lock and the MCS lock with traditional spin-loops exhibit a constant overhead. However, the T_{lock} measured with the MCS lock is lower than the one obtained with the ticket lock. This is due the classic advantage of MCS w.r.t other locking techniques [8]: spinning on locally-accessible flags is extremely convenient in terms of the inherent traffic on the interconnection structure to maintain the coherence of the flag lines. The T_{lock} values obtained by the MCS lock implemented using the `monitor/mwait` instructions are considerably higher, reflecting the result obtained in the first benchmark. However, such implementation should be taken into account in programs that heavily exploit Hyper-Threading, given that traditional busy-waiting techniques based on spin-loops are very resource consuming, as demonstrated in Section 4.1.

Tile64Pro. On the Tile64Pro we propose a MCS implementation that exploits the UDN network. Each record stores a DynamicHeader. During the lock acquisition, the busy-waiting is performed through a `UDN_receive()` operation, i.e. the thread waits until a special message is received. During the lock release, the holding thread passes the lock to its successor by reading the DynamicHeader from the corresponding record, and transmitting a one-word message with the `UDN_send()` operation. This technique produces a small network traffic also in presence of a large number of threads that contend for the lock (e.g. no invalidation message is produced to pass the lock to the successor thread). Atomic compare-and-swap and swap operations are emulated through the primitives available in the *Tilera Multicore Library* (**TMC**) [4].

The results are shown in Figure 7b. We have compared our MCS lock with the ticket lock version available on the TMC library. With a critical section of 5 μsec, T_{lock} remains constant by increasing the number of threads. However, with the MCS lock T_{lock} is half the contention time measured with the ticket lock. The experiments with a higher level of contention (i.e. a critical section of 0 μsec) emphasize the effectiveness of our implementation. For the ticket lock, T_{lock} slowly grows with the number of threads. On the other hand the MCS lock features a near-constant behavior, with a minimum peak near to 33 threads that requires a further investigation in the future.

Broadcom XLP. On the Broadcom XLP the MCS lock implementation is similar to the Tile64Pro. Each record stores a Header that identifies a specific NxCPU. The busy-waiting phase is implemented using a while-loop in which we test the result of the non-blocking `xlp_message_receive()`, and we release the processor resources through the `xlp_message_wait()` (see Section 4.1). The holding thread passes the lock by executing a `xlp_message_send()` to the NxCPU addressed by the header of the next record. Similarly to the Tile64Pro, the exploitation of the FMN network avoids to generate additional coherence traffic to pass the lock to the next thread, leading to a fast and efficient notification between NxCPUs. The atomic instructions required by the MCS algorithm have been emulated through the primitives available in the *HyperExec* Broadcom library [3].

From Figure 7c, the scalability of our MCS implementation is evident. For both the experiments with critical sections of 5 and 0 μsec, T_{lock} remains constant by increasing the number of threads. A different behavior is measured with the classic ticket lock version, in which the contention time increases with the number of threads (especially with a higher level of contention - i.e. with a critical section of 0 μsec). Moreover, as previously demonstrated, our MCS lock is a suitable alternative to synchronize threads executed on the four contexts of the same core.

5 Final discussion and Conclusions

In this paper, we studied two synchronization mechanisms: a *one-way notification* and a *locking* mechanism. We pointed out the importance of the busy-waiting phase to realize implementations that represent a proper trade-off between responsiveness and resource consumption. We showed how to exploit hardware facilities of three test-bed platforms to obtain alternative implementations.

For what concerns one-way notification, we highlighted the importance of T_{wake} as a measure of the mechanism responsiveness. On the Intel platform we showed that a plain spin-loop exhibits the best T_{wake}. However, in case multiple SMT contexts are used within the same core, a solution based upon the `pause` or (even better) `monitor/mwait` instructions is preferable for a smarter usage of the core resources. Unfortunately, the `monitor/mwait` instructions can be currently used in kernel mode only. A busy-waiting technique based upon a direct exchange of firmware messages was implemented on the Tile64Pro and the Broadcom XLP network processors. In the former case, a very low T_{wake} and a good scalability were obtained using the UDN network and a proper caching policy. For the latter case, despite of values of T_{wake} surprisingly higher than spin-loops, the FMN network is the best solution when the notification involves threads mapped onto the four SMT contexts of each core.

The results about different busy-waiting implementations were used to enhance a well-known locking algorithm (MCS lock). The busy-waiting phase can be performed using `monitor/mwait` instructions on Intel or, on network processors, their core-to-core networks. These MCS variants were compared with other traditional locking algorithms (e.g. notably a classic Ticket Lock). The results on the Intel platform confirm the high scalability of the MCS algorithm. However, the `monitor/mwait` instructions, introduced to hopefully lighten the busy-waiting cost, turn out to be ineffective until multiple SMT contexts are enabled. The same kind of experiments were repeated on the two network processors. Here, the alternative implementations of MCS lock exhibit attractive performance.

In the future we plan to extend this work by studying the implications of our results on other synchronization mechanisms, such as barriers and lock-free single-producer single-consumer queues.

References

[1] Nikos Anastopoulos and Nectarios Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–8. IEEE, 2008.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[3] BroadCom/Netlogic. Netlogic hyperexec," 2012, documentation from netlogic microsystems website http://www.netlogicmicro.com/.

[4] Tilera Corporation. Tilera application reference guide, 2012.

[5] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1st edition, 1998.

[6] Intel Corporation. *Intel$^{\circledR}$ 64 and IA-32 Architectures Software Developer's Manual*, 2012.

[7] Shaoshan Liu and Jean-Luc Gaudiot. Synchronization mechanisms on modern multi-core architectures. 4697:290–303, 2007. 10.1007/978-3-540-74309-5-28.

[8] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, 1991.

[9] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Efficient synchronization for embedded on-chip multiprocessors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(10):1049–1062, October 2006.

[10] Maeda Toshiyuki. Kernel mode linux: Toward an operating system protected by a type theory. In *Advances in Computing Science – ASIAN 2003. Programming Languages and Distributed Computation Programming Languages and Distributed Computation*. Springer Berlin Heidelberg, 2003.