

Efficient Synchronization for Embedded On-Chip Multiprocessors

Matteo Monchiero, *Student Member, IEEE*, Gianluca Palermo, *Member, IEEE*, Cristina Silvano, *Member, IEEE*, and Oreste Villa

Abstract—This paper investigates optimized synchronization techniques for shared memory on-chip multiprocessors (CMPs) based on network-on-chip (NoC) and targeted at future mobile systems. The proposed solution is based on the idea of locally performing synchronization operations requiring continuous polling of a shared variable, thus, featuring large contentions (e.g., spin locks and barriers). A hardware (HW) module, the synchronization-operation buffer (SB), has been introduced to queue and to manage the requests issued by the processors. By using this mechanism, we propose a spin lock implementation requiring a constant number of network transactions and memory accesses per lock acquisition. The SB also supports an efficient implementation of barriers. Experimental validation has been carried out by using GRAPES, a cycle-accurate performance/power simulation platform for multiprocessor systems-on-chip (MPSoCs). Two different architectures have been explored to prove that the proposed approach is effective independently from caches and coherence schemes adopted. For an eight-processor target architecture, we show that the SB-based solution achieves up to 50% performance improvement and 30% energy saving with respect to synchronization based on the caching of the synchronization variables and directory-based coherence protocol. Furthermore, we prove the scalability of the proposed approach when the number of processors increases.

Index Terms—Embedded systems, energy-aware systems, multi-processor system-on-chip (MPSoC), network-on-chip (NoC), synchronization.

I. INTRODUCTION

THE TRENDS for embedded systems impose multiple cores tightly interconnected and integrated on a single chip [1]–[3]. Among embedded devices, multiprocessor systems on-chip (MPSoCs) are emerging as appealing solutions for complex applications targeting future mobile multimedia systems [4]–[6]. MPSoCs are typically composed of processors, memories, and application specific coprocessors. They may include homogeneous on-chip multiprocessors (CMPs) or, alternatively, specialized processors. Communication is often provided by advanced interconnect, typically implemented with high performance and efficient network-on-chip (NoC) [7]–[9].

In general, applications running on MPSoCs require a high bandwidth memory subsystem, as well as an efficient thread synchronization mechanism. A careful design of the synchro-

nization has to be carried out, meeting tight design constraints in terms of performance, cost, and power consumption.

This paper focuses on power/performance optimization of the synchronization mechanisms for shared memory homogeneous CMPs based on the NoC approach. We introduce a shared memory controller architecture, specifically optimized to reduce the overhead of *busy-wait* synchronization algorithms. These techniques are based on the continuous polling of shared memory locations. They are the base of the most common synchronization routines, like mutual exclusions or barriers, and feature a large amount of memory and interconnect contention [10].

The idea is to store in the memory controller the requests issued by the processing elements, by means of a dedicated hardware block: the synchronization-operation buffer (SB). The SB *locally* manages the polling on shared locations, avoiding the network traffic and memory accesses otherwise needed by software-implemented synchronization. For example, considering spin locks, the SB holds the list of the contending threads, updating lock ownership when the contended lock is released. Concerning barriers, the SB avoids polling in a similar fashion, also making this common synchronization construct *busy-wait* free.

This paper extends our previous works [11], [12] in several aspects. We outline the importance of the joint optimization of barriers and locks, on the basis of the experimental characterization of the target application set. We show that barrier overhead dominates and, therefore, lock optimization [13] cannot solely be effective for many applications which heavily rely on barriers. Furthermore, we analyze the microarchitecture of the SB and we introduce an optimized version featuring better scalability and lower complexity.

The optimization of busy-wait synchronization proposed to date exploits the locality of cache memories, at the cost of maintaining the coherence for the data used for synchronization. Our solution is effective, both in systems with and without caches, avoiding the additional cost of coherence protocols on synchronization variables. The SB-based implementations of spin locks and barriers, respectively, feature $O(1)$ and $O(P)$ memory references and network transactions (where P is the number of processors).

The analysis and the exploration of the proposed synchronization technique have been carried out by using GRAPES [14], an MPSoC modelling and simulation platform. GRAPES features cycle-accurate simulation environment for power/performance evaluation of MPSoC architectures. Experimental results show that significant performance improvement and en-

Manuscript received October 20, 2005; revised March 16, 2006. This work was supported in part by STMicroelectronics.

The authors are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milan, Italy (e-mail: monchier@elet.polimi.it; gpalermo@elet.polimi.it; silvano@elet.polimi.it; ovilla@elet.polimi.it).

Digital Object Identifier 10.1109/TVLSI.2006.884147

ergy reduction can be achieved by SB-based architectures. We also prove the scalability of the proposed mechanism by evaluating performance and energy consumption of systems with 4, 8, and 16 processors.

The paper is organized as follows. Section II presents the background and the related works. The architecture of the SB has been discussed in Section III, while in Section IV, the implementation of the SB in two target MPSoC architectures has been presented. In Section V, the simulation setup and experimental results are presented, while in Section VI some concluding remarks are reported.

II. BACKGROUND AND RELATED WORK

Multiprocessor systems try to boost overall performance through parallel execution of different threads. In order to support these features, they embed hardware dedicated to inter-processor communication and synchronization. Programming models based on shared memory weak consistency models, relying on synchronization primitives are among the most popular ones [5], [15]–[17]. In this case, the programmer is provided with user-level software routines, which build on hardware-supplied synchronization instructions. These are typically implemented with uninterruptible read-Modify-write operations (e.g., *test-and-set*, *fetch-and-increment*) or with coupled *load-linked/store-conditional* instructions; for example, the IA32 *cmpxchg* and PowerPC *lwarx/stwxcx*. Atomic operations can be implemented as processor-centric operations or at the home node (memory controller) [18]. In our work, all the synchronization operations are handled actively by the memory controller.

Many optimizations of the synchronization constructs have been proposed in the literature. In [10], a survey of efficient software algorithms for spin locks and barriers is presented. Simple locks, implemented with *test-and-set* loops, can be optimized by delaying each consecutive probe of the lock. Different approaches using constant delay or proportional and exponential back-off can also be adopted. The *ticket lock* is an alternative spin lock implementation based on *fetch-and-increment* instruction and shared counters. However, these solutions generate large contention because of the polling of shared variables. One of the most efficient implementations of spin locks is the *queuing lock*, which is based on a shared structure holding the queue of the threads waiting for the lock. The hardware implementation of the queuing lock has been proposed in [13], [19], and [20] and features significantly lower overhead than software mechanisms. In systems with caches, queue-based locks feature $O(1)$ traffic, since once the lock address has been obtained, polling can be performed locally in the cache.

The most straightforward algorithms for barriers are the *centralized barriers*. Once any processor reaches the barrier, it updates a shared structure, which is monitored to check when all the processors arrive. Efficient implementations have also been proposed in [10], targeting large scale distributed systems. These mainly rely on tree structures. The basic idea is to represent a single shared variable, targeted by the polling, by means of a set of variables organized in a tree data structure, synchronizing the threads by groups starting from the tree leaves. Since

our target system consists of a small scale CMP, we do not consider these algorithms, which would result in a large overhead for a small number of processors. Alternatively, we propose an optimization of barriers, based on HW-implemented *event* primitives.

Several research groups have recently proposed a multiprocessor architecture and execution model based on transactional memory [21], [22]. This approach relies on advanced speculative hardware to support the atomic execution of software transactions.

Although transactional systems are promising for high-performance computers, we consider this kind of solution impractical for our target systems, since it requires complex and power-hungry hardware design. Furthermore it is unclear if the transactional programming model will cause an improvement in the software development phase, as compared with traditional lock-based programming [23].

In the embedded system field, MPSoCs are usually programmed with *ad hoc* techniques, and no parallel programming models are used. Issues of structured programming models and synchronization in MPSoCs have been recently addressed in [5], [17], and [24].

In [5], synchronization is provided by an *ad hoc* semaphore unit, and caching of semaphores is adopted to reduce synchronization contention. Cache coherence is enforced by using a modified version of the MESI write-invalidate snoopy coherence protocol.

Paulin *et al.* [17] face problems related to parallel programming model design in embedded multiprocessors. The authors propose a symmetric multi processing (SMP) programming model based on a hardware unit, the concurrency engine, and a software layer which exposes synchronization hardware to the programmer by means of a C++ application programming interface (API).

In [24], the authors present an evaluation of several cache coherence schemes. They explore three scenarios, in the context of a shared memory MPSoC based on bus interconnect. Snoop-based, software-based (no caching of shared data), and OS-based coherence are analyzed, showing that the OS-based solution has excessively high cost, and snoop-based coherence, though competitive in terms of performance, features significant power dissipation when coherence traffic grows.

This paper represents a further step in the study of MPSoCs, introducing an *ad hoc* architecture to support low-complexity and power-aware, yet efficient, synchronization.

III. SB

In this paper, we propose a hardware block known as the SB, which enhances the memory controller with efficient management of the synchronization operations requiring continuous polling of a shared variable. Synchronization primitives can cause significant memory and interconnect contentions. We consider the implementation of two primitives: *spin locks* and *events*, which can be used as basic blocks of most used software synchronization routines.

The basic idea, that we propose, is to *locally* manage spin locks and events, in the memory controller, maintaining a buffer which holds a queue of pending operations. The buffer logic also

```

acquire_lock(lock *L, int P){
    if (Test-and-set(L)){ /* Try to lock L */
        /* Notify to P */
        send_msg_back("lock acquired",P);
    }
    else {
        /* Add the new entry to the SB */
        insert_at_SB_tail(L,P);
    }
}
release_lock(lock *L){
    int P=-1;
    /* Search for the first contender in the SB */
    P = search_SB_for(L);
    if (P>0) {
        /* remove the entry and notify */
        retire_from_SB(L,P);
        send_msg_back("lock acquired",P);
    }
    else {
        *L=0;
    }
}

```

Fig. 1. Algorithms used by the SB to manage locks: lock acquire and lock release.

monitors for changes in the contended shared variable. In this way, polling can be avoided and the overhead due to contention is significantly reduced. In the following, we separately describe how the SB works for spin locks and events.

A. Spin Locks Management

Concerning spin locks, each entry of the SB corresponds to a specific lock and consists of two fields containing information related to the issuing processor and lock address. The SB is managed as a circular buffer, with the tail pointer indicating the most recent entry and the head pointer indicating the oldest one.

The SB content is modified whenever a request of *lock acquire* and *lock release* is received by the memory controller, according to the algorithm shown in Fig. 1 where the algorithms used to manage locks by the SB are shown. When the processor P wants to acquire the lock at address L , a *test-and-set* instruction tries to set the lock in the memory. If it does not succeed, an entry is allocated at SB tail, related to the pair $\{L, P\}$. On each request for lock release, the SB is searched for matching entries and if a processor (P) waiting for the lock (L) is found, P is notified that it has acquired the lock. If no processor is contending L , the lock is released. The algorithm serves spin locks in a first-in first-out (FIFO) order, avoiding starvation situations.

B. Events Management

The SB can also be extended to manage other operations, widely used in busy-wait synchronization constructs. *Events* are primitives which generate a signal when an “event” occurs. We consider events on shared variables and we specify an event with a pair $\{variable, event_value\}$. An event occurs when the content of *variable* is equal to *event_value*. Typical implementation of events occurs by means of polling of the monitored variable.

Events are used in barriers. Barrier implementation adopted in this paper is *sense-reversing centralized barrier* as shown in

```

event(int *variable, int event_value, int P){
    /* test the event */
    if (*variable == event_value){
        /* notify the event */
        send_msg_back("event",P);
    }
    else {
        /* Add the new entry to the SB */
        insert_at_SB_tail(variable, event_value, P);
    }
}
write(int *variable, int value){
    SB_entry_list entry_list=empty;
    *variable=value;
    /* look for all the contenders of the event */
    entry_list = search_SB_for(variable);
    for each entry in entry_list {
        /* check the event value */
        if (SB[entry].event_value == value) {
            /* notify the event */
            send_msg_back("event",entry);
            /* retire from the SB */
            retire_from_SB(variable,entry);
        }
    }
}

```

Fig. 2. Algorithms used by the SB to manage events: the event requests and the write on an event variable.

[15]. It is composed of a lock-protected critical section and an event to monitor if all the processors have reached it.

The algorithms used by the SB to manage events are shown in Fig. 2. The SB entry must hold the monitored variable address (*variable*), the value which triggers the signal (*event_value*) and the processor that requested the event (P). The SB is accessed when an event service is requested to the memory controller. If the monitored variable value is different from the event value, the triple $\{variable\ address, event\ value, requesting\ processor\}$ is inserted at SB tail. On each write to the shared memory, the SB is searched to test if any event is satisfied. If that happens, the corresponding SB entries are retired and the event is signaled to the processors which requested the event.

C. Performance Evaluation

Associative search in the SB is the most critical operation for performance. It is performed by temporal ordering, starting from the oldest entry (SB head) to the most recent one (SB tail). Retirement can happen out-of-order, since locks and events, when referring to different locations, are independent.

Table I shows performance of SB-based spin locks and events. We consider three different lock initial conditions: depending if the lock is unlocked in the memory, if it is locked and there is a single contender, or if multiple contenders exist. For events, two different initial conditions are analyzed: $event ==$ means that the event variable is equal to the event value, while $event! =$ means they are different. All the network transactions and memory references are taken into account, since there is no additional coherence overhead.

Even if the lock (or event) is stored in the memory (and not cached), the SB needs only one memory access and at most 2 (or $1 + P$) transactions, where P is the number of processors. This is the minimum overhead to satisfy functional requirements of

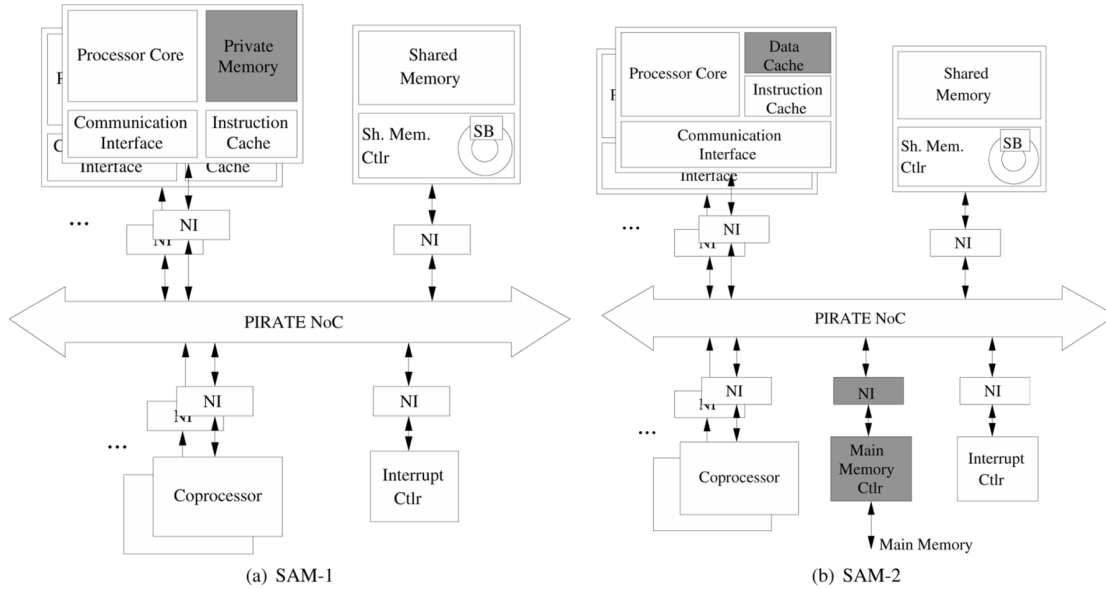


Fig. 3. Architecture of the two target systems. Differences between the two systems are emphasized by using gray boxes. (a) SAM-1. (b) SAM-2.

TABLE I
SB THEORETICAL PERFORMANCE FOR LOCK ACQUISITION AND EVENT SIGNAL GENERATION (WHERE P IS THE NUMBER OF PROCESSORS)

	Lock idle in memory	Locked, single contender	Locked, N contenders	Event ==	Event !=
Number of memory references	1	1	1	1	1
Number of network transactions	2	2	2	2	1+P

the synchronization. In any case, caching and related coherence are not necessary for the SB, since caching of synchronization variables is implicit in its structure. Cache-based mechanisms (such as in [19]) require at least the same number of memory accesses, and achieve single transaction performance only due to inter-cache messaging.

The number of network transactions required by using the SB to acquire a lock is constant, independent of the number of processors. Regarding events, each time a signal is generated, the amount of network transactions is linear with the number of contending processors, which is the complexity that dominates the barrier algorithm used in this paper. When we consider a request for an event, i.e., a call to *event(...)*, a constant number of network transactions is required. The number of memory accesses is constant for spin locks and events.

SB-based primitives can be used regardless of whether the system includes cache memories. In cache-free multiprocessors, SB is effective to prevent large amounts of network traffic due to polling. In cache-based multiprocessors, the SB provides sequential consistency on synchronization variables without the need of caching (and coherency support) for those variables.

Furthermore, the SB can be extended to deal with multithreaded processing elements (PEs). In this case, each thread can be a contender for a lock or for any other synchronization variable. This basically has an impact on the size of the SB, which must potentially store $P \times T$ requests (where P is the number of processors and T is the number of threads per processor). In any case, the algorithms shown in this section remain substantially unchanged. In this paper, we consider single-threaded PEs, except in Section IV-B, where implementation details of the SB are discussed even considering multithreaded PEs.

IV. IMPLEMENTATION

In this section, we discuss the implementation of the SB in a MPSoC architecture based on NoC interconnect. The characteristics of the target architecture and the HW/SW layer designed to support parallel programming are discussed in Section IV-A. The proposed SB implementation is presented in Section IV-B and analyzed in Section IV-C.

A. Target System Architecture

In this paper, we consider two different MPSoC architectures (Fig. 3): SAM-1 and SAM-2. SAM stands for shared-memory ARM multiprocessor, since the processor cores are implemented with ARM processors. The former integrates private memory on each PE, while the latter has off-chip private memory (hosted in the main memory) and data caches in each PE. The differences between the two architectures are emphasized in Fig. 3 by the gray boxes.

1) *Overview:* Fig. 3(a) shows the block diagram of the system architecture of SAM-1. It is composed of several PEs, each of them including a processor core, private memory, and instruction cache. The processor is a five-stage pipeline core based on the StrongARM microprocessor.¹ The communication interface provides the interface with the channel, i.e., the PIRATE NoC [9], that is a configurable and scalable high bandwidth on-chip interconnect. Network interface (NI) modules supply a translation layer between NoC protocol and the IPs composing the system. Memory modules can be distributed across the NoC, providing on-chip shared memory space for data. Furthermore, custom coprocessors supplying specialized

¹[Online]. Available: <http://sourceforge.net/projects/simit-arm/>

functions can be easily plugged into the system. Asynchronous events are managed by the interrupt controller.

The PE of SAM-2 [the architecture is shown in Fig. 3(b)] includes a data cache, while private memory is mapped on the external main memory. Interface with the main memory is provided by the main memory controller.

2) *Memory Model*: The memory model of SAM-1 and SAM-2 is composed of separate private memory space and shared memory space. Private memory space exists for each PE and cannot be shown by any other PE in the system except for its owner. Shared portions of the addressing space are used for synchronization and data exchange.

Temporary view of the memory space (both private and shared) is provided only in SAM-2 by cache memories in each PE. We assume a weak consistency model [16], which requires sequential consistency for the synchronization variables, while memory operations on data between two synchronization points can be reordered. The coherence of cached data is assured by flushing shared cache lines at each synchronization point. Consistency of the synchronization variables can be enforced by using different mechanisms (no caching, caching with coherence protocol, queuing locks and the SB have been analyzed in Section V).

Globally, the main difference between the two architectures is that SAM-1 does not cache shared data, while SAM-2 caches shared data.

3) *Interconnect*: The interconnect is the key element of the multiprocessor system, since it provides a low latency communication layer, capable of minimizing the overhead due to thread spawning and synchronization. This approach, when compared to bus-based solutions, solves physical limitations due to wire latency, providing higher bandwidth and parallelism in the communication.

PIRATE [9] is a NoC composed of a set of parameterizable interconnection elements (routers) connected by using different topologies. The router architecture is based on a crossbar topology, where the $N \times N$ network connects the N input FIFO queues with N output FIFO queues. The number of each input (output) FIFO queue is configurable as well as queue length. The incoming packets are stored in the corresponding input queue, then forwarded to the destination output queue. The I/O queues and the crossbar are controlled by the router controller, that includes the logic to implement routing, arbitration and virtual-channels. The PIRATE router is a three-stage pipelined architecture that implements table-based routing and wormhole switching. The PIRATE NI implements the adaptation between the asynchronous protocol of the network and the specific protocol adopted by each IP-node composing the system. The NI is responsible of the packeting of the service requests coming from the IP nodes. The service level is best effort and there is a round robin priority based scheduling mechanism.

4) *Programming Model*: Low-level application programming interface (API) has been provided to support the parallel programming model based on shared memory. In addition to primitives which atomically read, modify, and write a memory location (based on the *test-and-set* instruction), primitives to support SB operations (such as *acquire_lock*, *release_lock*, and *event*) have been provided. The target architecture supports

fork/join parallel execution model. Thread spawning capabilities have been supplied through the *create_thread* primitive, while the *wait_for_end* primitive has been used to implement joint semantics.

We built PARMACS macros [25] on the top of the API to fully support parallel programming. The porting has been carried out by using the implementation of the PARMACS for SGI [25] as a guideline. Based on these PARMACS macros, the usage of the SB is completely hidden for the software programmer.

B. SB

This section describes the SB implementation and the exploration of the SB design space. We first summarize the SB organization as proposed in our previous work, then we discuss an enhancement of the SB which improves scalability and performance.

In [12], we presented the SB as an independent module communicating with the memory controller through a dedicated bus, while in [11], the SB is embedded in the memory controller itself. In both works, we have proposed a split microarchitecture for the SB, consisting of two separate buffers: the *Events Buffer* (EB), to manage the *events*, and the *Locks Buffer* (LB), dedicated to the *spin locks*. The LB-entry stores the address referenced by a *spin lock* operation and the pair PE-thread that makes the request, while the EB-entry has three fields: the address of the monitored variable, the *event* value and the requesting pair PE-thread.

The main problems of the microarchitecture presented in [11], [12] are due to scalability and complexity.

- Even if for small multiprocessors the SB size is reasonably small, a large number of cores and threads requires a fairly large SB. The size of both the EB and the LB is determined by the maximum number of contending threads that, in a multithreaded CMP, is equal to $T \times P$ (where T is the number of thread per processor and P is the number of processors). This fact has an impact on the size of the associative search, making it potentially slow and power hungry.
- In the previous SB organization, the address identifiers (both for locks and events) are not unique, therefore, it is necessary that associative search matches multiple entries, and that establishes an ordering (only for the LB). Considering events, on each write, the list of processors associated with the event must be retrieved, while for locks, on each release, the oldest contender must be found making the management even more complex.

Fig. 4 shows the enhanced version of the SB microarchitecture proposed in this paper, aiming at solving these two problems. The main ideas are that the EB and the LB are integrated in a single structure where unique identifiers for the synchronization variables are stored.

The integration of the EB and LB in a single structure does not increase the size of the associative search, since a system cannot have more than $T \times P$ parallel requests for locks and events (where T is the number of threads per processor and P is the number of processors). Since locks and events need different structures, each SB entry holds an *event*-value field. This is not taken into consideration when a lock is managed.

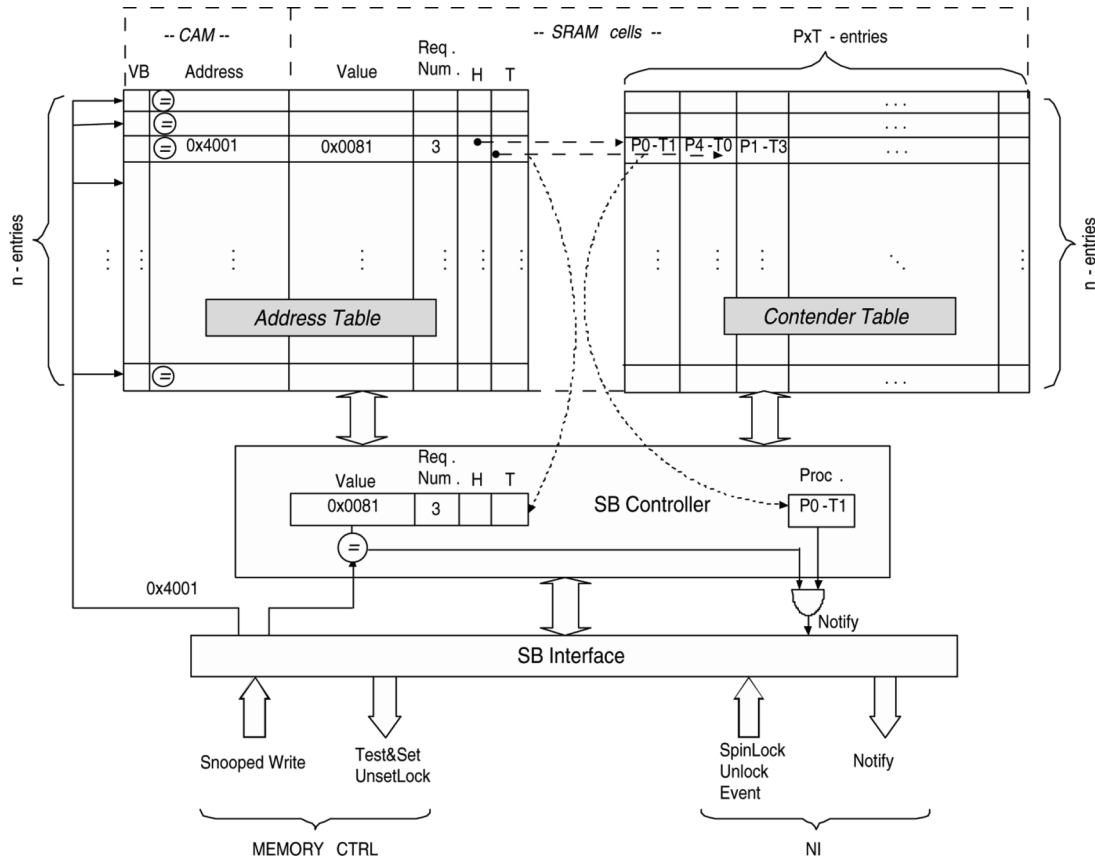


Fig. 4. Proposed enhanced version of the SB.

The management of a unique identifier for each synchronization variable has two main advantages. First, the number of valid entries in the SB is equal to the number of active synchronization variables, while for the original SB, it would be equal to the number of active requests. Second, the FIFO management of the lock requests is now done explicitly in the RAM, reducing the cost of the temporal ordering with respect to our previous implementation [11], [12].

The microarchitecture proposed in Fig. 4 is composed of two tables: the *address table* and the *contender table*. In the *address table*, the CAM holds the identifiers of events and locks which are currently active. In the CAM, multiple requests to the same variable collapse in the same entry. For example, considering a barrier based on an event primitive, only a single entry related to the event is present in the CAM. Each CAM entry is used to index a structure composed of four fields: the *event-value* field, the number of active requests on the considered variable and *head* (H) and *tail* (T) fields. (H) and (T) are used to manage the list of contenders for the synchronization variable stored in the *contender table*. The number of rows and columns of the *contender table* are, respectively, dependent on the number of entries in the *address table* and on the maximum number of possible contending threads ($P \times T$, where P is the number of processors and T is the number of threads per processor).²

The proposed enhanced microarchitecture needs a larger number of memory cells with respect to the microarchitecture

²When the number of active synchronization variables is higher than the number of SB entries, a polling mechanism must be activated.

presented in [11] and [12], but solves the problem of the FIFO management for the locks, avoiding the logic for the temporal ordering.

Most of the applications running on embedded MPSoC are barrier-synchronization dominated, so most of the requests to the SB are related to a single synchronization variable. In the latter case, the CAM of the enhanced SB can be designed with a small number of entries, which does not increase with the number of cores in the system. In fact, the CAM size is given by the maximum number of lock/event addresses which are simultaneously active. For the parallel applications used in our experimental results (mostly composed of homogeneous parallel threads synchronized at barriers), all the threads work on less than four synchronization variables at the same time.

C. SB Microarchitecture Evaluation

Table II shows the performance in terms of area, latency, and energy of the enhanced SB microarchitecture with respect to the original one [11], [12]. The values have been obtained by using an architectural model derived from [27], [28], and [29]. The values presented in Table II are reported by varying the maximum number of concurrent requests ($T \times P$) to the SB. The table has been split in two parts: the first one considers the original version of the SB, while the second is related to the enhanced version. In the second part of the table, the performance of the SB by varying the number of the CAM entries has been explored. The number of CAM entries corresponds to the maximum number of synchronization variables concurrently active.

TABLE II
SB MICROARCHITECTURES EVALUATION

	# Concurrent requests ($T \times P$)					
	2	4	8	16	32	64
Original SB [12] [11]						
Latency [ps]	100.91	118.10	176.53	389.55	1200.32	4360.70
Area [μm^2]	650.7	1309.2	2634.2	5299.8	10662.4	21450.2
Energy [pJ]	0.54	1.08	2.18	4.37	8.78	17.64
Enhanced SB						
2-entries						
Latency [ps]	96.89	97.66	99.19	102.25	108.38	120.63
Area [μm^2]	584.1	615.4	678.2	803.6	1054.5	1556.2
Energy [pJ]	0.50	0.52	0.56	0.63	0.78	1.07
4-entries						
Latency [ps]	-	111.59	114.66	120.78	133.03	157.53
Area [μm^2]	-	1105.1	1168.6	1293.0	1544.8	2046.4
Energy [pJ]	-	0.97	1.00	1.08	1.22	1.51
8-entries						
Latency [ps]	-	-	169.64	181.89	206.38	255.38
Area [μm^2]	-	-	2148.7	2273.6	2524.2	3026.6
Energy [pJ]	-	-	1.89	1.97	2.11	2.41
16-entries						
Latency [ps]	-	-	-	400.27	449.27	547.27
Area [μm^2]	-	-	-	4233.4	4484.1	4986.9
Energy [pJ]	-	-	-	3.75	3.90	4.19
32-entries						
Latency [ps]	-	-	-	-	1319.76	1515.75
Area [μm^2]	-	-	-	-	8404.5	8906.2
Energy [pJ]	-	-	-	-	7.47	7.76
64-entries						
Latency [ps]	-	-	-	-	-	4991.57
Area [μm^2]	-	-	-	-	-	16746.3
Energy [pJ]	-	-	-	-	-	14.90

This part of the table is an upper triangular matrix, since it is meaningless to consider more CAM entries with respect to the maximum number of possible concurrent requests.

Increasing the maximum number of concurrent requests, the latency, area, and energy trends are similar for the original SB and for the enhanced one, if the number of CAM entries is equal to the maximum value $T \times P$ (see the main diagonal in Table II for the enhanced SB). Nevertheless, for area and energy, the enhanced SB scales slightly better with respect to the original one, since the logic to manage the temporal ordering is no longer necessary. On the other hand, the delay for the enhanced SB does not scale as well as compared with the original one, since it is necessary to access the *contender table*.

Considering the enhanced SB microarchitecture with a fixed number of CAM entries, it is important to note that, for all the considered metrics (latency, area, and energy), only a small overhead exists to manage more concurrent requests. This is due to the fact that only the *contender table* is scaled up to support the increasing number of concurrent requests. This characteristic is very useful for designing systems that execute applications with the maximum number of synchronization variables simultaneously active, for instance as in embedded MPSoCs.

In the experimental results shown in Section V, we examined systems composed of 4, 8, and 16 processors using the enhanced version of the SB with 4, 8, and 16 concurrent synchronization requests and a 4-entry CAM (since all the considered applications do not have more than 4 synchronization variables active at the same time). To evaluate the cost of SB design alternatives, a 512-kB SRAM memory generated by CACTI [28] can be used, featuring 12.97 mm² for area, 1.52 ns as access time, and 0.6 nJ of energy per access. If we compare the SB in the worst case of 16 concurrent requests (as reported in Table II) to the selected

TABLE III
PROGRAM BENCHMARKS

Benchmark	Description
<i>FFT</i>	Complex 1D radix- \sqrt{n} 6-step FFT [33]
<i>LU-1</i>	Lower/upper triangular matrix factorization [33]
<i>LU-2</i>	<i>idem</i> , but optimized for spatial locality (<i>contiguous blocks</i>) [33]
<i>Radix</i>	Integer radix sort [33]
<i>Bio</i>	Multiagent negotiation algorithm for a heart-pacing simulation and control [34]
<i>Chebyshev</i>	Chebyshev series parallel evaluation [35]
<i>15-puzzle</i>	Calculation of the 15-puzzle game solution based on tree movements exploration [36]
<i>QWST</i>	Quarter Wave Sine Transform [37]
<i>Norm</i>	Parallel vector normalization
<i>Matrix</i>	Matrix multiplication

SRAM, the corresponding values of energy per access and area are far less than 1%.

The impact of the SB on the delay in a memory controller running at 200 MHz is at most 3% of the clock cycle. This ensures that SB operations can be performed within a single clock cycle of the memory controller.

V. EXPERIMENTAL RESULTS

In this section, we discuss experimental results obtained by using the GRAPES simulation framework, providing evaluation of the SB-based MPSoC architecture. In particular, Section V-A describes the simulation setup and Section V-B shows the impact of the SB on NoC traffic. In Section V-C, results obtained on the simulated benchmarks for the eight-core system are presented. Finally, Section V-D discusses the SB impact when scaling the system size.

A. Simulation Setup

The GRAPES framework is a cycle-based simulation platform to simulate MPSoCs based on complex interconnect and IPs [14]. The NoC is modeled by using transaction level cycle-accurate SystemC description. IP cores, i.e., PEs, memories, and coprocessors, are described as independent modules connected to the NoC. PEs are modeled by using the SimIt ARM simulator [30] and cycle-accurate memory models have been developed according to the modelling technique outlined in [31].

The system-level power model is provided in the GRAPES framework. The model accounts for processors, memories, and NoC power consumption. Power model parameters, as well as architectural parameters, have been estimated by gate-level simulation [9] (with 90-nm STMicroelectronics CMOS technology) or by using publicly available models (from [32] for ARM cores and CACTI for memories [28]). The accuracy of our power model is directly derived from the accuracy of the models used for each component [9], [28], [32].

Table III lists the simulated benchmarks with a short description of each one. We selected some kernels from the SPLASH-2 benchmark suite [33]. A reduced working set has been used to fit constraints imposed by our target architecture, while providing significant complexity. Furthermore, we simulated several applications from different domains (biomedicine, numerical analysis, artificial intelligence, and DSP). These benchmarks have been parallelized by following the approach used in the SPLASH-2 suite [33].

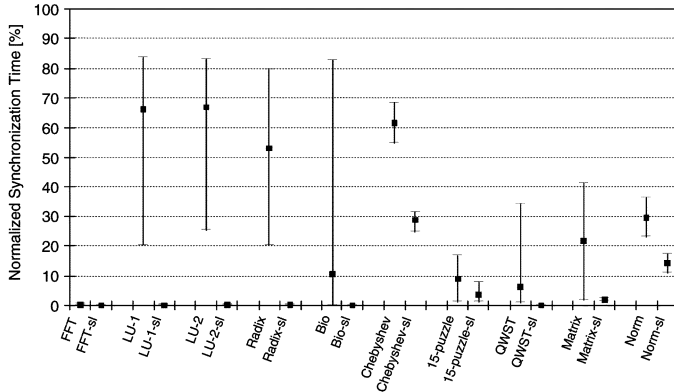


Fig. 5. Normalized synchronization time due to spin locks and events ($\langle \text{Bench} \rangle$) and due to spin locks only ($\langle \text{Bench} \rangle - \text{sl}$) for each benchmark execution time. Each bar represents average (black square), minimum (lower line), and maximum (upper line) values computed over all the processors in eight-PEs SAM-1 architecture.

TABLE IV
ARCHITECTURE PARAMETERS

Parameter	Value
PE Clock Freq.	200 MHz
NoC Clock Freq.	400 MHz
NoC Datapath Width	32 bit
Memory Controller Clock Freq.	200 MHz
Shared Memory Size	512KB
Data cache (only for SAM-2)	8KB 4-way 32B block write-back
Instruction Cache	16KB 4-way 32B block write-back
Number of PEs	4, 8, 16
NoC Topology	Mesh

To characterize the selected set of benchmarks in terms of the percentage of synchronization, Fig. 5 shows the normalized synchronization time due to spin locks and events (see $\langle \text{Bench} \rangle$) and due to spin locks only (see $\langle \text{Bench} \rangle - \text{sl}$) for each benchmark execution time. In particular, each bar represents the average (black square), the minimum (lower line), and the maximum (upper line) values computed over all the processors in eight-PEs SAM-1 architecture. Synchronization overhead is significant for all the benchmarks apart from FFT, *15-puzzle*, and QWST. It can account for up to 68% of the execution time for *LU-2* (on average), and it may feature significant variability across different processors. It is important to observe that most of the synchronization time is due to events. Spin lock contribution is not appreciable, apart from *Chebyshev* and *Norm*.

Table IV shows details of the base architecture configurations used for simulation. PEs have been clocked at 200 MHz as well as shared memories interfaces and control logic. PIRATE NoC can run up to 850 MHz, but we set network clock at 400 MHz to minimize latency while not dramatically affecting the total power. NoC topology has been configured as a square mesh with $(n \times \text{PE} + 2)$ for SAM-1, while $(n \times \text{PE} + 3)$ for SAM-2. Three base configurations of the system have been simulated (comprising 4, 8, or 16 PEs) and 512-kB shared memory.

In the following, we briefly describe the synchronization schemes evaluated on the target architectures (SAM-1 and SAM2).

The following schemes to manage synchronization have been explored for SAM-1.

- 1) MEM: All the shared variables, including synchronization variables, are not cached, thus, they reside in the shared

memory. For this architecture, we adopt software implementation of spin locks and barriers based on fixed delay polling and *test-and-set* instruction. The execution of the *test-and-set* instruction is performed in the shared memory controller.

- 2) Hardware Queuing Lock (HwQL): All the locks are managed through the HwQL mechanism, while events are implemented in software through a polling-based routine. The implementation of the HwQL has been derived from [13]. It is a centralized queuing lock implementation in the shared memory controller.
- 3) SB: The SB is implemented in the shared memory controller to assure sequential consistency of synchronization variables.

To explore the efficacy of the SB for the SAM-2 architecture, we simulated the following schemes to enforce ordering on synchronization data accesses.

- 1) MEM: Synchronization variables are not cached, thus, they reside in the shared memory. This configuration implements synchronization by means of software spin locks (*test-and-set*-based) and software events, featuring fixed-delay polling. The execution of the *test-and-set* instruction is performed in the shared memory controller.
- 2) HwQL: As for SAM-1, locks are managed by the HwQL in the shared memory controller. Events are implemented by a software routine, and they are not cached.
- 3) SB: The SB is implemented in the shared memory controller to assure sequential consistency of synchronization variables.
- 4) COH: Synchronization data are cached and a coherency protocol for synchronization data is used. The write-invalidate directory-based protocol [15] has been implemented, suitable for the SAM-2 target architecture, that is based on a network interconnect. This configuration features optimized software implementation of locks and events ($O(1)$ memory references and network transactions), based on local polling in the caches.

B. NoC Traffic Analysis

In this subsection, we evaluate how the SB modifies the NoC traffic behavior. Among the set of simulated benchmarks, results are reported in Figs. 6 and 7 for the *Norm* and *Bio* applications. These two applications represent two typical behaviors of the bandwidth due to the synchronization: burst and uniform. Burst and uniform behavior can be, respectively, identified in the middle plots of Fig. 6(a) for *Norm* and Fig. 7(a) for *Bio* referring to the synchronization bandwidth.

Figs. 6 and 7 show the total aggregate bandwidth (top), the synchronization aggregate bandwidth (middle), and the maximum packet latency (bottom) for the NoC in system configurations (a) without the SB and (b) with SB for the *Norm* and *Bio* benchmarks executed by the SAM-1 architecture including eight-PEs.³ Each point in both figures has been computed based on a mobile time-window of 15 μs .

³Similar results have been obtained for SAM-2 architecture, but they are not shown in the paper.

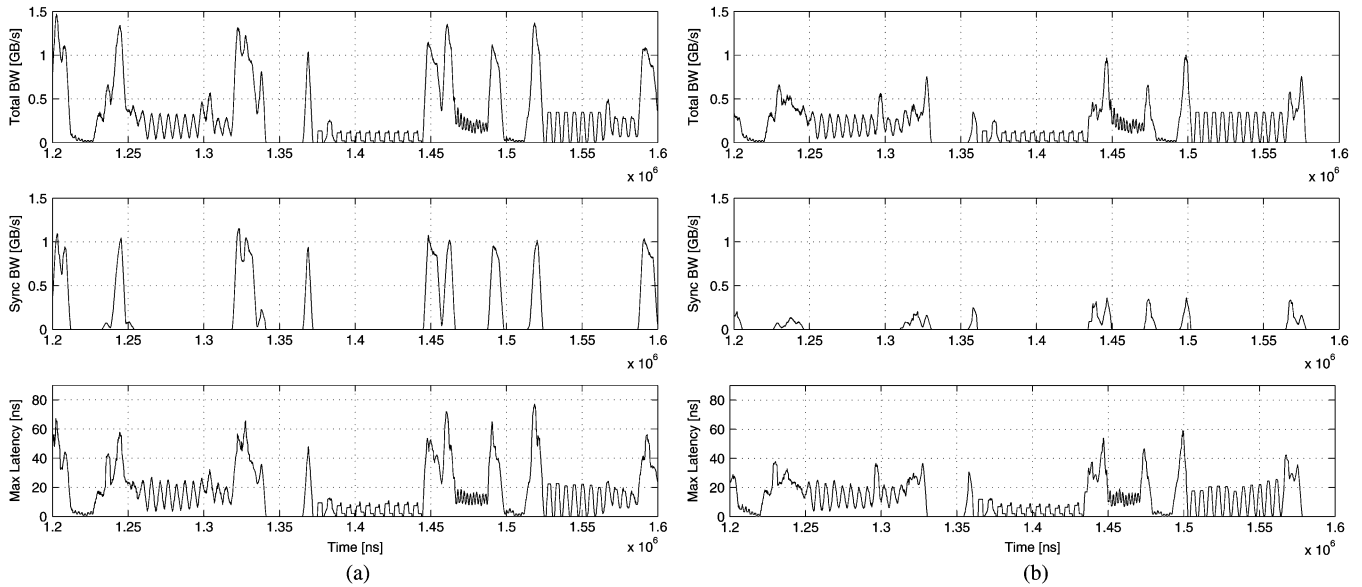


Fig. 6. NoC total aggregate bandwidth (top), synchronization aggregate bandwidth (middle), and maximum packet latency (bottom) for eight-core SAM-1 system (a) without the SB and (b) with SB for the *Norm* benchmark.

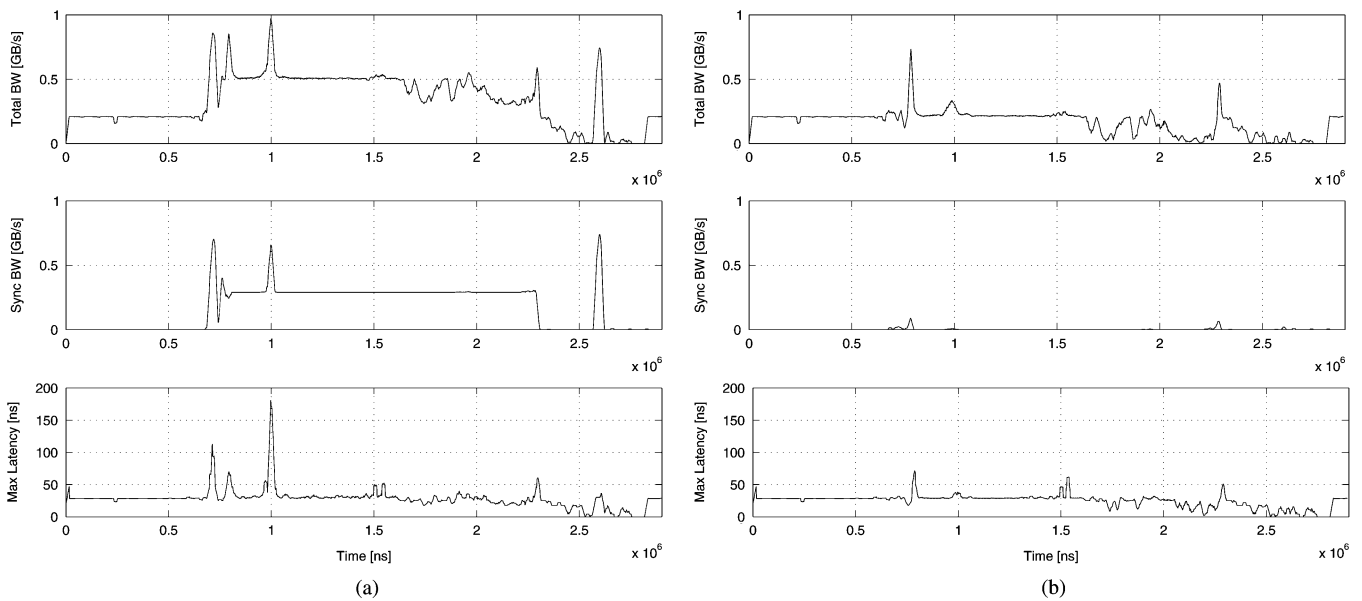


Fig. 7. NoC aggregate bandwidth (top), synchronization aggregate bandwidth (middle), and maximum packet latency (bottom) for eight-core SAM-1 system without (a) the SB and (b) with SB for the *Bio* benchmark.

The *aggregate bandwidth* for the NoC has been defined as the sum of the bandwidth (data rate) over all network links (including router-to-router and router-to-node links). The *total aggregate bandwidth* is composed of instructions, private data, shared data, and synchronization data, while the *synchronization aggregate bandwidth* is due to synchronization data only. The *maximum packet latency* for the NoC has been defined as the maximum latency to deliver one packet from source to destination network-interfaces.

Fig. 6 reports the results for the *Norm* benchmark over a time interval from 1.2×10^6 ns to 1.6×10^6 ns. The *Norm* benchmark is composed of three parallel loops: the first one is the parallel initialization of a vector of *double*, the second one performs search of the maximum value, the third one normalizes the vector elements. Barriers are used to synchronize the parallel loops, and spin locks rule data accesses. The three parallel

loops of the benchmark can be respectively recognized from the profile of Fig. 6(a) from 1.24×10^6 ns to 1.33×10^6 ns, from 1.37×10^6 ns to 1.45×10^6 ns and from 1.52×10^6 ns to 1.59×10^6 ns.

In Fig. 6(a), all the bandwidth peaks, which can be observed in the middle plot (synchronization aggregate bandwidth), have been filtered by the SB. These peaks are related to synchronization points (spin locks and barriers). A similar behavior can also be observed for the maximum packet latency: indeed, latency peaks decrease when SB has been introduced.

Fig. 7 reports the results for the *Bio* benchmark over a time interval from 0 to 2.9×10^6 ns. The *Bio* benchmark is based on a multiagent negotiation algorithm [34] featuring continuous synchronization. In fact, the synchronization traffic profile presents a few peaks (0.6×10^6 ns, 1×10^6 ns, and 2.5×10^6 ns) and a plateau ranging from 0.6×10^6 ns to 2.5×10^6 ns. The

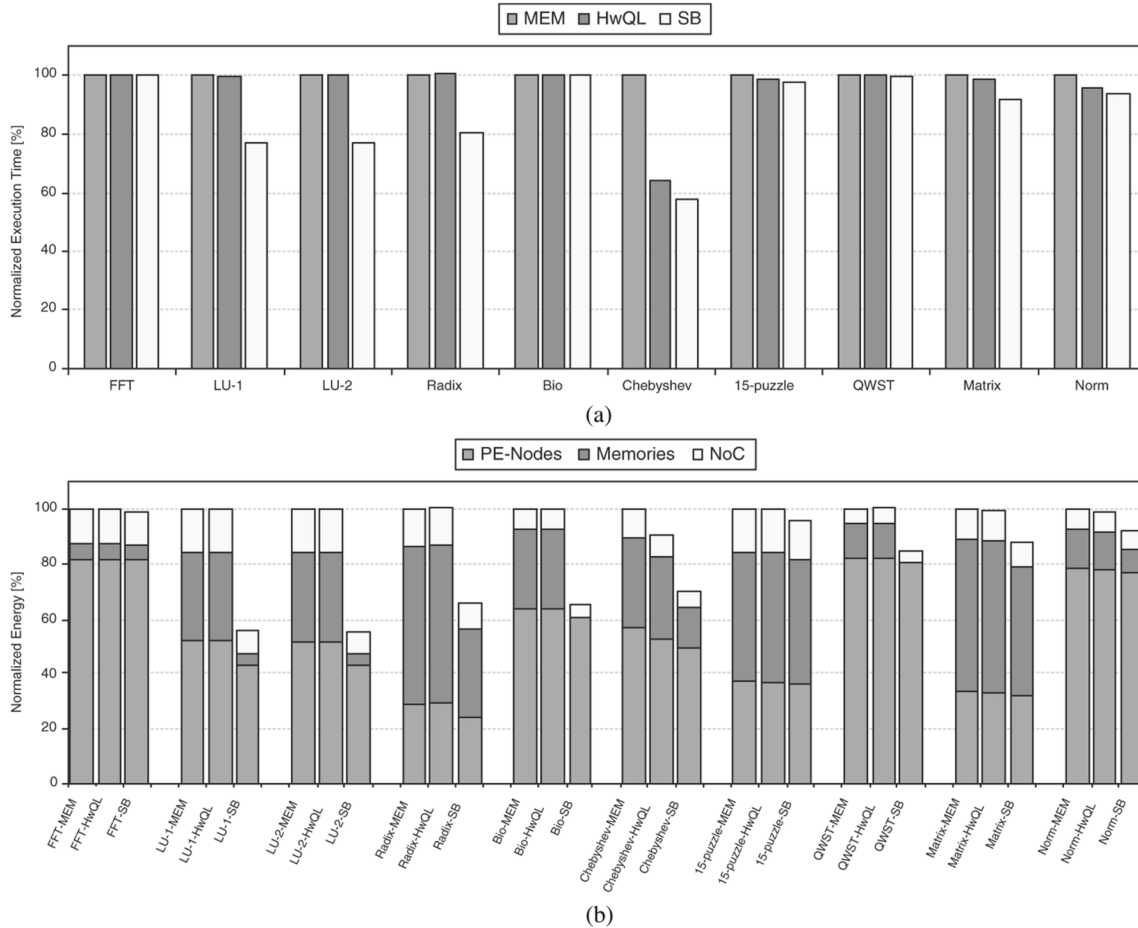


Fig. 8. Performance and energy evaluation for eight-core SAM-1. The comparison is between the base architecture with synchronization variables residing in the shared memory (*MEM*), the architecture based on HwQL and on SB. (a) Performance. (b) Energy.

traffic peaks and the plateau in Fig. 7(b) have been significantly reduced by the SB. Furthermore, some spikes in the maximum packet latency have been almost eliminated (e.g., see at 1×10^6 ns).

Globally, the filtering effects on the synchronization aggregate bandwidth, for both burst and uniform behaviors, are due to the introduction of the SB, which locally manages spin locks and events, thus, avoiding useless synchronization traffic.

C. System Energy and Performance Evaluation

This subsection discusses the experimental evaluation of the SB-based architectures from the joint point of view of performance and energy efficiency. First, the results for SAM-1 architecture have been reported, then the results for SAM-2.

In Fig. 8, the impacts of the SB on the overall system performance and energy consumption have been evaluated, for each simulated benchmark, considering the eight-core SAM-1 architecture. Experimental values obtained for HwQL and SB have been normalized with respect to the MEM scheme.

Performance improvement up to 25% has been obtained for *LU-1*, *LU-2*, and *Radix*; up to 40% for *Chebyshev*, and up to 10% for *Matrix* and *Norm*. The other benchmarks show only limited benefits.

Load balance characteristics are important in determining how much the SB impacts the system behavior. If the load

is unbalanced, busy-wait synchronization forces some PEs to wait. If spinning synchronization operations are software implemented, PEs do a lot of polling during these time slots. In contrast, as the SB solution does not require polling, the execution is faster.

As shown in Fig. 5, LUs, *Radix* and *Chebyshev* spend a lot of time in synchronization (65%, 50%, and 60% of the execution time on average), featuring significant load unbalance. Due to this reason, for some benchmarks (*FFT*, *15-puzzle*, and *QWST*), the impact of the SB is negligible, while, for synchronization-intensive benchmarks (such as LUs, *Radix*, and *Chebyshev*), the SB significantly improves performance.

Since the time spent for synchronization due to locks is negligible for most of the selected benchmarks (except for *Chebyshev* and *Norm*), the effect of the introduction of the hardware queuing lock is not so evident, unlike the introduction of the SB. For *Chebyshev* and *Norm* benchmarks, the HwQL and the SB behave similarly (the SB is slightly better), since the synchronization due to locks is significant.

Fig. 8(b) shows energy breakdown for each benchmark. We consider three components of the total energy: the PE energy component, which includes the whole node energy consumption, including memory (SAM-1) or caches (SAM-2); the memories component which refers to shared memory modules; the NoC component, related to interconnect energy consumption.

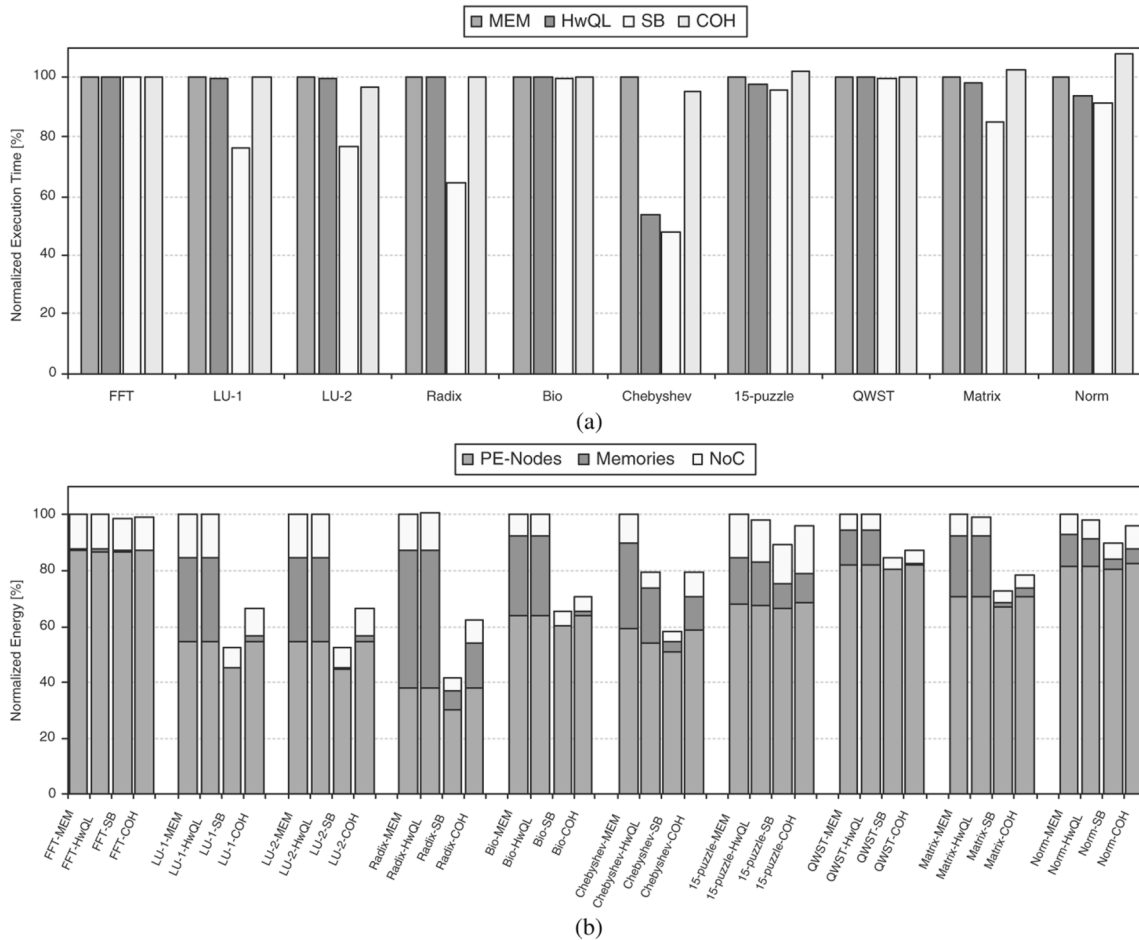


Fig. 9. Performance and energy evaluation for eight-core SAM-2. The comparison is among the architecture with synchronization variables residing in the shared memory (MEM), the architecture based on the HwQL, on the SB and on the COH. (a) Performance. (b) Energy.

The SB contribution for the energy results has not been shown separately, but included into the memory bar, since the SB bar would have been unreadable (the same consideration applies for HwQL).

As shown in Fig. 8(b), energy saving is achieved in the NoC, in the memories and in the PEs, for the LUs, *Radix* and *Chebyshev* (45%–35%). As already observed, these are the benchmarks featuring larger synchronization overhead and, thus, the SB effects are more evident. The SB avoids the generation of useless synchronization traffic and also reduces memory accesses. In addition, the SB makes PEs save energy, since it allows processor cores to switch into a low-power state, while waiting for lock release. This cannot happen if the spin locks are implemented in software, because the processor must poll the locked location until the lock is released (the same considerations apply for events).

For *Bio*, *15-puzzle*, and *QWST*, the SB has little effect on the performance. In any case, as observed in Fig. 7 for *Bio*, the SB reduces NoC traffic and memory accesses. These three benchmarks spend a large fraction of the execution time in computation, so the SB effects cannot be directly observed from system performance analysis.

In Fig. 9, performance and energy evaluations are reported for the four schemes considered for SAM-2: MEM, HwQL, SB, and COH.

In Fig. 9(a), when synchronization variables are cached and cache coherency protocol is enabled (COH), no advantage can be observed with respect to MEM: even if the polling is local in the caches, coherency protocol overhead compensates. In particular, the coherency overhead is evident for *15-puzzle*, *matrix*, and *Norm*, causing performance decreases.

The SB-based configuration improves performance for a subset of the benchmarks. For the LUs, the SB reduces the execution time of the 25% with respect to MEM, while for *Radix* and *Chebyshev* SB, respectively, features 40% and 55% reduction.

SB-based synchronization does not suffer from coherency protocol overhead, resulting in better performance than COH: 25% improvement for LUs, 40% for *Radix*, 50% for *Chebyshev* and up to 15% for the other benchmarks.

Fig. 9(b) reports energy breakdown for each benchmark and synchronization scheme. Caching synchronization variables (COH) is effective in reducing the contention due to polling. In fact, it significantly reduces memory energy, affecting those benchmarks with large synchronization overhead. *LU-1*, *LU-2*, *Radix*, *Chebyshev*, feature energy reduction up to 40% with respect to MEM configuration. An energy reduction up to 20% is also obtained for those benchmarks where the coherency overhead implies a performance reduction. On the other hand, caching synchronization variables means that coherency protocol has to be enforced to guarantee consistency.

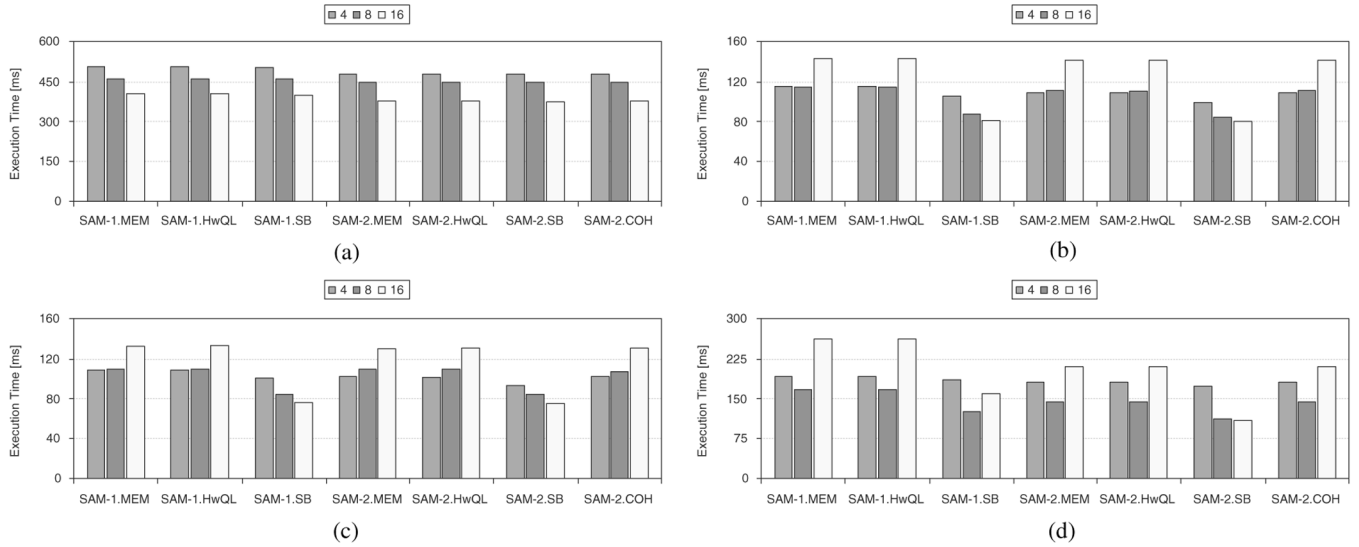


Fig. 10. Execution time scaling trends for the simulated SPLASH-2 kernels by varying the number of processors for different architectures. (a) FFT. (b) *LU-1*. (c) *LU-2*. (d) *Radix*.

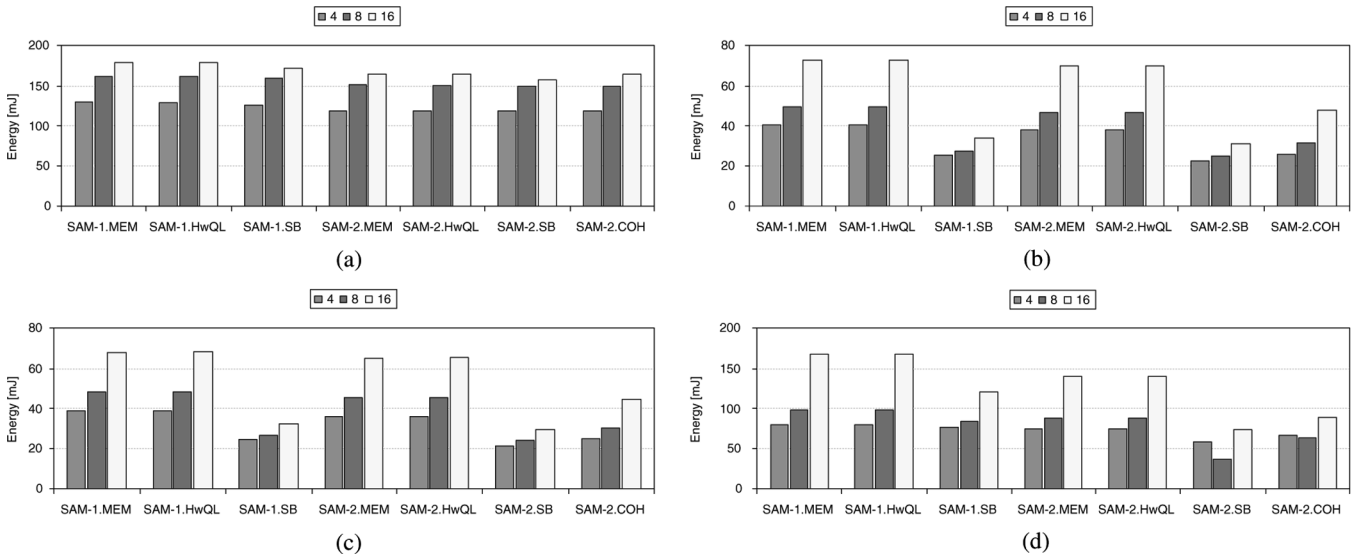


Fig. 11. Energy scaling trends for the simulated SPLASH-2 kernels by varying the number of processors for different architectures. (a) FFT. (b) *LU-1*. (c) *LU-2*. (d) *Radix*.

The SB behaves in a similar way, but it avoids the overhead of coherency protocol, resulting in 20%–40% energy reduction with respect to COH configuration for the LUs, *Radix*, *Chebyshev* and up to 10% for the other benchmarks by reducing all the energy components. In fact, network transactions due to coherency are avoided and performance increase of the SB-based solution impacts on the PE energy.

Concerning HwQL, considerations similar to those applied to SAM-1 can be applied to SAM-2. The efficacy of the HwQL is bound by the amount of lock synchronization. For lock intensive applications, the HwQL mechanism is as effective as the SB, while for the others it does not provide any appreciable advantage.

D. Scaling the System Size

Scaling trends for both performance and energy are presented, in this subsection, for each SPLASH-2 kernel (namely, FFT, LU-1, LU-2, and Radix) and systems including 4, 8, and 16 PEs. The focus of this subsection is to evaluate the effects

of the SB on scaling. It is beyond the scope of this paper to discuss application scaling in a multiprocessor environment. This topic has been already examined in the literature. In particular, scaling issues of SPLASH-2 benchmarks have been analyzed in [38] and [39]. Referring to the experiments shown in [38] (conducted on the SGI Origin2000 machine), all the applications considered by the authors (which include the FFT and *Radix*) do not scale for a large number of processors, with the original problem size. Since we consider a reduced working set and relatively small shared memory, this problem is emphasized for our target system.

As shown in Fig. 10, the efficacy of the scaling depends on the application and the synchronization scheme. It can be seen that FFT scales independently of the caching schemes and the SB does not impact the scaling. As previously observed, this benchmark is characterized by small load unbalance.

For *LU-1*, *LU-2*, and *Radix* benchmarks is evident that the SB makes more effective scaling. This is true both for performance (Fig. 10) and for energy (Fig. 11). *SAM-1.MEM* and

SAM-2.MEM configurations do not scale well since synchronization variables reside in the memory and, when polling is required, longer network transactions are needed as system size increases. Similar behavior can be noted for *SAM-1.HwQL* and *SAM-2.HwQL*.

Scaling the number of processors, the synchronization overhead due only to locks does not change a lot with respect to the values shown in Fig. 5. This implies that *HwQL* optimizes only a small fraction of the execution time. The *SAM-2.COH* configuration relies on coherence protocol to avoid the polling, but this increases the traffic as the system size increases.

VI. CONCLUSION

This paper explores energy/performance optimization techniques of busy-wait synchronization for on-chip multiprocessors based on NoCs. We suggest a memory module architecture, optimized to manage synchronization operations, by means of a dedicated hardware unit: the SB. The proposed mechanism features low overhead synchronization, by locally managing synchronization in the memory controller. This solution has low-complexity, since it avoids the additional cost of the coherence protocol, which is often used by state-of-the-art synchronization solutions.

Experimental results have been carried out by integrating the SB in the GRAPES MPSoC platform. The results have shown that SB features up to 50% of performance improvement and 30% of energy reduction with respect to synchronization techniques based on caching of the synchronization variables and directory-based coherence protocol. The SB is effective especially on those applications featuring significant synchronization overhead. As the system is scaled up, this phenomenon becomes more evident.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive and useful comments and suggestions. The authors would also like to thank R. Zafalon of STMicroelectronics for encouraging this research. Finally, many thanks to E. Coffey for checking the final version of this paper.

REFERENCES

- [1] R. Kalla, B. Sinharoy, and J. Tendler, "IBM Power5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, Mar./Apr. 2004.
- [2] C. McNairy and R. Bhatia, "Montecito: A dual-core, dual-thread Itanium processor," *IEEE Micro*, vol. 25, no. 2, pp. 10–20, Mar./Apr. 2005.
- [3] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded spare processor," *IEEE Micro*, pp. 21–29, Mar./Apr. 2005.
- [4] W. Wolf, "The future of multiprocessor systems-on-chip," in *Proc. Des. Autom. Conf.*, 2004, pp. 681–685.
- [5] B. Ackland, A. Anesko, D. Brinthaup, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O'Neill, J. Othmer, E. Skinger, K. Singh, J. Sweet, and C. Terman, "A single-chip 1.6 billion 16-b MAC/s multiprocessor DSP," *IEEE J. Solid-State Circuits*, vol. 35, no. 3, pp. 412–424, Mar. 2000.
- [6] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Des. Test Comput.*, vol. 18, no. 5, pp. 21–31, Sept./Oct. 2001.
- [7] L. Benini and G. D. Micheli, "Networks on chip: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.
- [8] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. Des. Autom. Conf.*, 2001, pp. 684–689.
- [9] G. Palermo and C. Silvano, "PIRATE: A framework for power/performance exploration of network-on-chip architectures," in *Proc. PATMOS-04: Proc. Int. Workshop Power Timing Modeling, Optimization Simulation*, 2004, pp. 521–531.
- [10] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Jan. 1991.
- [11] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "An efficient synchronization technique for multiprocessor systems on-chip," in *Proc. MEDEA'05—Int. Workshop Memory Performance: Dealing With Appl., Syst. Arch.*, 2005, pp. 33–40.
- [12] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Power/performance hardware optimization for synchronization intensive applications in MPSoCs," in *Proc. DATE06, Des., Autom. Test Eur.*, 2006.
- [13] T. Anderson, "The performance of spin lock alternatives for shared memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.
- [14] V. Catalano, M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "GRAPES: A modular simulation framework for MPSoC," Dept. Elect. Inf., Politecnico di Milano, Milano, Italy, Internal 14Report 2006.01, 2006.
- [15] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2003.
- [16] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," Western Research Laboratories, CA, Tech. Rep. 95/7, 1995.
- [17] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Proc. CODES-ISSS*, 2004, pp. 48–53.
- [18] L. Zhang, Z. Fang, and J. Carter, "Highly efficient synchronization based on active memory operations," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2004, p. 58a.
- [19] A. Kagi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat QOLB," in *Proc. 24th Ann. Int. Symp. Comput. Arch. (ISCA)*, 1997, pp. 170–180.
- [20] D. Lenoski, J. Laudon, K. Gharachorloo, W.-I. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," *IEEE Comput.*, vol. 25, no. 3, pp. 63–79, 1992.
- [21] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proc. 10th Int. Conf. Arch. Support Program. Lang. Operat. Syst. (ASPLOS-X)*, 2002, pp. 5–17.
- [22] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with transactional coherence and consistency (TCC)," in *Proc. 11th Int. Conf. Arch. Support Program. Lang. Operat. Syst. (ASPLOS-XI)*, 2004, pp. 1–13.
- [23] C. Blundell and C. L. M. Martin, "Deconstructing transactional semantics: The subtleties of atomicity," in *Proc. Ann. Workshop Duplicating, Deconstructing, Debunking (WDDD)*, 2005.
- [24] M. Loghi and M. Poncino, "Exploring energy/performance tradeoffs in shared memory MPSoC: Snoop-based cache coherence vs. software solutions," in *Proc. DATE*, 2005, pp. 508–513.
- [25] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra, "Implementing PARMACS macros for shared-memory multiprocessor environment," Dept. Comput. Arch., Polytechnic Univ. Catalunya, Spain, Tech. Rep. UPC-DAC-1997-07, 1997.
- [26] J. Baxter, "Stanford Parallel Applications for Shared Memory (SPLASH)," 2001 [Online]. Available: <http://www-flash.stanford.edu/apps/SPLASH/>
- [27] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. 27th Ann. Int. Symp. Comput. Arch.*, 2000, pp. 83–94, 2000.
- [28] P. Shivakumar and N. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," HP Western Research Laboratories, 2001.
- [29] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. ISCA*, 1997, pp. 206–218.
- [30] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *Proc. DATE*, 2003, p. 10556.
- [31] O. Villa, P. Schaumont, I. Verbauwhede, M. Monchiero, and G. Palermo, "Fast dynamic memory integration in co-simulation frameworks for multiprocessor system on-chip," in *Proc. DATE*, 2005, pp. 804–805.
- [32] A. Sinha, N. Ickes, and A. Chandrakasan, "Instruction level and operating system profiling for energy exposed software," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 6, pp. 1044–1057, Dec. 2003.

- [33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. ISCA*, 1995, pp. 24–36 [Online]. Available: citeseer.csail.mit.edu/woo95splash.html
- [34] A. Beda, N. Gatti, and F. Amigoni, "Heart-rate pacing simulation and control via multiagent systems," in *Proc. Workshop Agents Applied Health Care Eur. Conf. Artif. Intel. (ECAI)*, 2004, pp. 22–30.
- [35] R. Barrio and J. Sabadell, "A parallel algorithm to evaluate chebyshev series on a message passing environment," *SIAM J. Scientific Comput.*, vol. 20, no. 3, pp. 964–969, 1999.
- [36] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. San Mateo, CA: Morgan Kaufmann, 2002.
- [37] S. Clarke, "Netlib Library" 2006 [Online]. Available: <http://www.netlib.org/vfftpack/>
- [38] D. Jiang and J. P. Singh, "Scaling application performance on a cache-coherent multiprocessor," in *Proc. ISCA*, 1999, pp. 305–306.
- [39] D. Jiang, B. O'Kelley, X. Yu, S. Kumar, A. Bilas, and J. P. Singh, "Application scaling under shared virtual memory on a cluster of snips," in *Proc. ICS*, 1999, pp. 165–174.



Gianluca Palermo (M'06) received the Laurea degree in electronic engineering and the Ph.D. degree in computer science from Politecnico di Milano, Milano, Italy, in 2002 and 2006, respectively.

He is currently an Assistant Researcher at the Department of Electronic Engineering and Computer Science, Politecnico di Milano. Previously, he was a Consultant Engineer in the Low Power Design Group of AST—STMicroelectronics, Agrate, Italy, where he worked on networks on-chip. His research interests include design methodologies and architectures for embedded systems, focusing on power estimation, on-chip multiprocessor, and networks on-chip.



Cristina Silvano (M'99) received the Laurea degree in electronic engineering from Politecnico di Milano, Milano, Italy, in 1987, and the Ph.D. degree in computer engineering from the University of Brescia, Italy, in 1999.

She is currently an Associate Professor at the Department of Electronic Engineering and Computer Science, Politecnico di Milano. From 1987 to 1996, she held the position of Senior Design Engineer at R&D Labs, of Bull HN Information Systems, Pregnana, Italy. From 2000 to 2002, she was an Assistant Professor at the Department of Computer Science, University of Milano. Her primary research interests are in the area of computer architectures and computer-aided design of digital systems, with particular emphasis on low-power design and systems-on-chip.



Matteo Monchiero (S'00) received the Laurea degree (*cum laude*) in electronic engineering from Politecnico di Milano, Milan, Italy, in 2003. He is currently pursuing the Ph.D. degree at Politecnico di Milano, working on the power/performance analysis and optimization of multicore architectures.

His main research interests include the area of computer architecture, with particular emphasis on multicore architectures, branch prediction, and low-power/thermal-aware microarchitectures. He is a Member of Technical Committee on Computer

Architecture (TCCA) and TC on Microprogramming and Microarchitecture (TCUARCH).



Oreste Villa received the Laurea degree in electronic engineering from the University of Cagliari, Cagliari, Italy, in 2003, and the M.S. degree in embedded systems design at the ALaRI Institute, Lugano, Switzerland, in 2004. He is currently pursuing the Ph.D. degree at Politecnico di Milano, Italy, majoring in design methodologies of multiprocessor architectures for embedded systems, focusing on power estimation and on-chip communication.

He is currently a Ph.D. Intern in the Applied Computer Science Group at Pacific Northwest National Laboratory (PNNL) in Richland, VA, where he is conducting research on operating systems for clustered multiprocessor architectures.