Power/Performance Hardware Optimization for Synchronization Intensive Applications in MPSoCs

Matteo Monchiero   Gianluca Palermo   Cristina Silvano   Oreste Villa

Politecnico di Milano – Milano, Italy

{monchier, gpalermo, silvano, ovilla}@elet.polimi.it

Abstract

This paper explores optimization techniques of the synchronization mechanisms for MPSoCs based on complex interconnect (Network-on-Chip), targeted at future power-efficient systems. The proposed solution is based on the idea of locally performing synchronization operations which require the continuous polling of a shared variable, thus featuring large contention (e.g. spin locks). We introduce a HW module, the Synchronization-operation Buffer (SB), which queues and manages the requests issued by the processors. Experimental validation has been carried out by using GRAPES, a cycle-accurate performance/power simulation platform. For 8-processor target architecture, we show that the proposed solution achieves up to 40% performance improvement and 30% energy saving with respect to synchronization based on directory-based coherence protocol.

1. Introduction

Many semiconductor firms are recently proposing systems, based on few cores, tightly interconnected and integrated on a single chip [12, 10]. Among embedded devices, MultiProcessor Systems on-Chip (MPSoCs) are emerging as appealing solutions for complex applications targeting future mobile systems [21, 3, 8].

Applications running on MPSoCs require high bandwidth memory subsystem, as well as efficient thread synchronization mechanisms. Careful design of the synchronization has to be carried on, meeting strict design constraints in terms of performance, as well as cost and power consumption. In particular energy efficiency of such systems, impacting on battery lifetime, has emerged as one of the main design issues [14].

The focus of this paper is on the optimization of synchronization mechanisms for shared memory MPSoCs, adopting the joint point of view of energy efficiency and performance optimization. *Busy-wait* techniques, which are the base of most common routines (e.g. mutual exclusion or barrier synchronization), feature large amount of memory and interconnect contention [13], generating useless activity in the system. In this paper, we introduce a hardware architecture, specifically optimized to reduce the overhead of the synchronization algorithms, which are based on the continuous polling of a shared memory location (e.g. spin locks and barriers). The idea is to locally manage the requests issued by the processing elements composing the system, by

means of a dedicated hardware block: the Synchronization-operation Buffer (SB). The SB effectively avoids traffic network and memory accesses, otherwise needed by software synchronization.

The optimization of busy-wait synchronization, so far proposed, exploits the locality of cache memories, at the cost of maintaining cache coherence on the data used for synchronization. Our solution is theoretically effective, both in systems with caches and without them, avoiding the additional cost of coherence protocols. Spin lock implementation, based on the SB, features $O(1)$ memory references and network transactions. It is scalable with the number of processing cores, needing resource budget which is linear with the number of processors of the system.

Analysis and exploration of the proposed architecture have been performed by using GRAPES, a NoC-based MPSoC platform for cycle-based power/performance evaluation. Experimental results show that significant performance improvement and energy reduction can be achieved by SB-based architecture. We prove the scalability of the proposed mechanism by evaluating performance and energy consumption of 4, 8 and 16 processors system.

The paper is organized as follows. Section 2 presents some related works. The architecture of the SB is discussed in Section 3. In Section 4, the target architecture is introduced. The implementation of the SB is presented in Section 5. Finally, in Section 6, details about simulation setup and experimental results are presented.

2. Related Work

Many optimizations of the synchronization constructs have been proposed in the literature. In [13] is presented a survey of efficient software algorithms for spin locks and barriers. 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. Several hardware implementations of queuing lock have been proposed (e.g. in [5, 9]), and it has been shown that they feature significant 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.

In the embedded system field, MPSoCs, typically composed of few heterogeneous cores, have been programmed with *ad hoc* techniques, while no parallel programming models have been used. Issues of structured programming models and synchronization in MPSoCs have been only recently

faced in [3, 17, 11]. In [3], synchronization is provided by *ad hoc* semaphore unit, and caching of semaphores is adopted to reduce synchronization contention. Paulin *et al.* [17] face problems related to parallel programming model design in embedded multiprocessors. The authors propose a SMP programming model based on a hardware unit, the Concurrency Engine and a software layer, which exposes to the programmer synchronization hardware by means of a C++ API. In [11] the authors present power/performance evaluation of several cache coherence schemes. They explore three scenarios (snoop-based, software-based and OS-based), in the context of a shared memory MPSoC based on bus interconnect.

For what concerns high performance multiprocessors, Rajwar and Goodman have recently proposed multiprocessor architecture and execution model based on transactional memory [19]. This approach relies on advanced speculative hardware to support the atomic execution of software transactions. We consider this kind of solution not suitable for our target systems, since it requires complex and power-hungry hardware design.

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

3. Synchronization-operation Buffer

In this paper, we propose a hardware block, the Synchronization-operation Buffer (SB), which enhances the shared memory architecture with optimal management of the synchronization operations requiring continuous polling of a shared variable. These are primitives which may be source of significant contention in the memory and in the interconnect. We consider the implementation of two primitives: spin locks and *events*, which can be used as basic blocks of the most used software synchronization routines.

The basic idea, that we propose, is to locally manage in the memory controller spin locks and events, maintaining a buffer which holds a queue of pending operations. In the buffer it is also monitored when the value of the contended shared variable changes. In this way, polling can be avoided and the overhead due to contention is significantly reduced. In the following, we describe separately how the SB works for spin locks and events. The SB differs from previously proposed lock queuing mechanisms, since it provides unified optimization of lock and barriers, and it decouples synchronization optimization from cache coherence issues.

For what concerns 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 a head pointer indicating the most recent entry and a tail 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 illustrated in Figure 1. 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 acquired the lock. If no processor is contending L, the lock is released.

```
acquire_lock(lock *L, int P){
        if (Test-and-set(L))
                send_msg_back("lock acquired",P);
        }
        else {
                insert_at_SB_tail(L,P);
        }
}
release_lock(lock *L){
        int P=-1;
        P = search_SB_for(L);
        if (P>0) {
                retire_from_SB(L,P);
                send_msg_back("lock acquired",P);
        }
        else {
                *L=0;
        }
}
```

Figure 1. Algorithm to manage the SB, on each lock acquire and lock release request

The SB can be extended to manage also 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 is by means of polling of the monitored variable. They can be used in barriers implementation to monitor if all the processors have reached it.

The algorithm we use to manage the SB for events is shown in Figure 2. The SB entry must hold the monitored variable address (*variable*), the value which triggers the signal (*event_value*) and which processor 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 it happens, the corresponding SB entries are retired and the event is signaled to the processors which requested the event.

Associative search in the SB is performed observing temporal ordering, starting from the oldest entry (SB head) to the most recent one (SB tail). However retirement can happen out-of-order, since locks and events, when referencing to different locations, are independent. The algorithm makes spin locks be served in FIFO order, making sure that starvation situations are avoided.

Network transactions required to acquire a lock are a constant number, 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. When we consider a request for an event, i.e. a call to *event(...)*, a constant number of network transactions is required. Memory accesses number is constant for spin locks and for events. For the events we leave out the write operations, needed to update the value of the monitored value in the memory. SB optimal size depends on the number of concurrent issues of spin locks and events. In a system composed of P processors, it is linear with P, consider-

```
event(int *variable, int event_value, int P){
        if (*variable == event_value)
                send_msg_back("event",P);
        }
        else {
                insert_at_SB_tail(variable, event_value, P);
        }
}
write(int *variable, int value){
        SB_entry_list entry_list=empty;
        *variable=value;
        entry_list = search_SB_for(variable);
        for each entry in entry_list {
                if (SB[entry].event_value == value) {
                        send_msg_back("event",entry);
                        retire_from_SB(variable,entry);
                }
        }
}
```

Figure 2. Algorithm to manage the SB for event primitives

Figure 3. Target system architecture

ing that each processor can issue only one spin lock/event at time. Otherwise the number of SB entries depends on $P \times N$, where N is the number of parallel issues which a single processor can manage. For example, this is the case of processors running multiple threads.

In cache-coherent systems the SB provides sequential consistency on synchronization variables without the requirement of a coherence protocol.

4. Target Architecture

In this section, we present the architecture of our target system and we introduce the HW/SW layer designed to support parallel programming.

Figure 3 shows the block diagram of the system architecture. It is composed of several Processing Elements (PEs), each of them comprising a Processor Core, Data and Instruction Cache. The Communication Interface provides the interface with the channel, i.e. the NoC. Communication layer is provided by the PIRATE NoC [15], a configurable and scalable high bandwidth interconnect. Network Interface (NI) modules supply a translation layer between NoC protocol and the IPs composing the system: processors, memories, coprocessors. Memory modules can be placed across the NoC, providing on-chip shared memory space for data. Interface with the main memory is provided by the Main Memory Controller. The SB is plugged into the system as a separate module communicating over the network. A dedicated link between the SB and the Shared Memory Controller exists to permit SB to/from shared memory operations. Furthermore custom coprocessors, supplying specialized functions, can be easily plugged in the system. Asynchronous events are managed by the Interrupt Controller.

The memory model is composed of separate private memory space and shared memory space. Private memory space exists for each PE and can't be seen by any other PE in the system except for its owner. Shared portions of the addressing space are used for synchronization and data exchange. While private memory maps to the off-chip Main Memory, shared memory is implemented by means of on-chip RAM.

Temporary view of the memory space (both the private and shared one) is provided by cache memories in each PE.
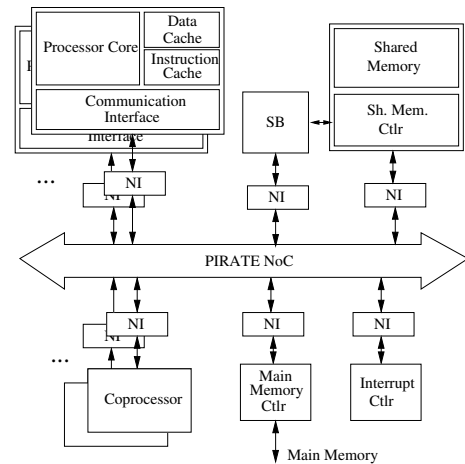
We assume a weak consistency model [4], which requires sequential consistency on synchronization variables, while memory operations on data, between two synchronization points, can be reordered. Both private and shared data are cached. The coherence of cached data is assured by flushing shared cache lines at each synchronization point. Consistency of the synchronization variables can be enforced by using different mechanisms (e.g. the SB), which are introduced in Section 6.

Low-level API is provided to support parallel programming model based on shared memory and fork/join parallel execution model. We built PARMACS macros [6] on the top of the API to fully support parallel programming. The porting has been carried out using as guideline the implementation of the PARMACS for SGI [1].

5. Synchronization-operation Buffer Implementation

The SB module has been implemented in the target system as stand-alone IP (see Figure 3), communicating over the interconnect with the cores, while a dedicated link provides the interface with the shared memory controller. We partitioned the SB core logic into two separate components. One is devoted to manage *events*, named *Events Buffer* (EB), the other one is dedicated to *spin locks*, named *Locks Buffer* (LB). In this way, the complexity of the associative search logic is divided, resulting in more efficient implementation. Furthermore, the structure of LB entries and EB entries are different. Each LB entry stores the address referenced by a *spin lock* operation (32 bits) and the PE that makes the request (8 bits), while the EB entry has three fields: the address of the monitored variable (32 bits), the *event* value (32 bits) and the requesting PE number (8 bits). Since address fields are used as tags for matching logic, they have been implemented in CAM, while other fields are in RAM.

In Table 1, synthesis results obtained with Synopsys Design Compiler, targeting 90 nm STMicroelectronics CMOS technology, are presented. Data for area, delay and power are normalized to area occupation, cycle time, and average access power of a 512KB on chip SRAM memory. Area for the

# entries	8	16	32	64
Cell area [%]	1.80	3.37	7.2	14.3
AS critical path [%]	23	27.4	22.2	23.2
AS power [%]	0.4	0.8	1.72	3.4

Table 1. Synthesis results for different size SBs

entire SB is shown for 4 different sizes (8, 16, 32 and 64 entries) to use in 4, 8, 16 and 32 PEs system (one LB and EB for each PE). These configurations comprise both the LB and the EB, which are equally sized. We report worst case power and critical path latency of the associative search phase (*AS*, *Associative Search* in Table 1). This is the most critical operation, which defines the minimum achievable clock cycle and features maximum power dissipation.

As expected, area and power grow linearly with SB size. Delay is substantially constant with size. In fact, critical path does not depend on the device size and long wire effects are not evident for small sizes. Constraints on the clock frequency (200MHz) have been imposed and are satisfied. Both power and area of the SB represent a small overhead with respect to the on chip memory for small scale systems.

When a lock is acquired by a processor, waiting in the SB, two concurrent operations must be performed: retirement from SB and sending back a message to the processor (it will take 30-40 NoC cycles to reach the processor, as measured with 8-core system). So the cost of acquiring a lock is mainly determined by the latency of a single network transaction.

6. Experimental Results

In this section, we discuss experimental results obtained by using the GRAPES simulation framework, providing evaluation of SB-based MPSoC architectures.

The GRAPES framework is based on the SystemC simulation kernel and a cycle-accurate SystemC-OCCN description of the NoC. 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 [18], while cycle-accurate memory models have been developed from scratch.

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 [15] (with 90 nm STMicroelectronics CMOS technology) or extracted by experimental data (from [20] for the ARM cores and STMicroelectronics data for memories [2]). Accuracy of our power-model is directly derived from the accuracy of the models used for each component [15, 20, 2].

Table 2 lists the simulated benchmarks with a short description of each one. We selected some kernels from the Splash 2 benchmark suite [22]. 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, games, DSP).

Table 3 shows details of the architecture base configurations used for simulation. PEs have been clocked at 200

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

Table 2. Program Benchmarks

Parameter	Value
PE clock frequency	200 MHz
NoC clock frequency	400 MHz
Memory controller clock frequency	200 MHz
SB clock frequency	200 MHz
Shared memory size	512KB
Data cache	8KB 4-way 32B block, write-back
Instruction cache	16KB 4-way 32B block, write-back
#PE	4, 8, 16
NoC topology	Mesh

Table 3. Architecture parameters

MHz as well as shared memory interfaces and the SB. PI-RATE NoC can run up to 850 MHz, but we set network clock at 400 MHz to minimize latency, while not dramatically impacting on total power. NoC topology has been configured as a Mesh. Three base configurations of the system have been simulated (comprising 4, 8 or 16 PEs) and 512KB shared memory.

In order to explore the efficacy of the SB, we simulated the following schemes to enforce ordering on synchronization data accesses:

1. *MEM*. Synchronization variables are not cached, thus they reside in memory. This configuration implements synchronization by means of software spin locks (*test-and-set* based) and events, featuring fixed-delay polling [13].

2. *SB*. As Configuration 1 but, the SB is used. As previously stated, the SB assure sequential consistency on synchronization variables.

3. *COH*. Also synchronization data are cached and coherence protocol for synchronization data is used to maintain ordering. We choose to implement a simple write-invalidate directory-based protocol [16], suitable for our target architecture based on a network interconnect. This configuration features optimized lock primitives ($O(1)$ memory references and network transactions), based on local polling in the caches.

Figure 4 shows the instantaneous aggregate bandwidth of the NoC and the maximum packet latency, during the execution of the *Bio* benchmark, for 8 PEs system. *Bio* is based on multiagent negotiation algorithm [7] featuring continuous synchronization.

It can be observed that the traffic in the time interval ranging from $4 \times 10^6 ns$ to $11 \times 10^6 ns$ is significantly reduced by the SB: 0.3 GB/s without SB to 0.1 GB/s with SB. Also some long latency spikes are nearly eliminated (e.g. at $5 \times 10^6 ns$).
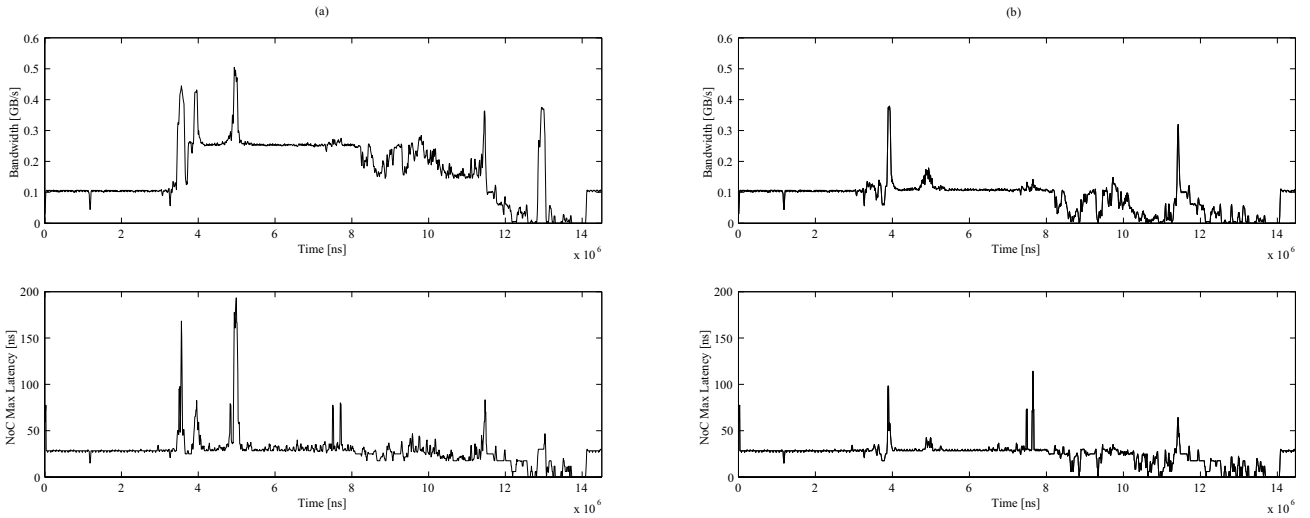
Figure 4. NoC aggregate bandwidth (top) and maximum packet latency (bottom) for systems without the SB (a) and with SB (b), for the *Bio* benchmark
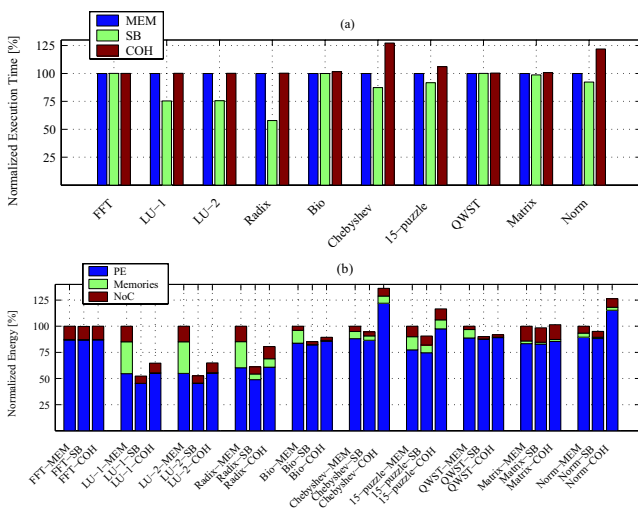

Figure 5. Performance (a) and energy (b) evaluation of different caching and synchronization schemes for 8 PEs systems

In Figure 5 the impact of the SB on the overall system performance and energy consumption is evaluated. Experimental values are normalized with respect to Configuration 1 (*MEM*).

As can be seen in Figure 5(a), when synchronization variables are cached and cache coherence protocol is enabled (*COH*), no advantage can be observed with respect to *MEM*: even if the polling is local in the caches, coherence protocol overhead compensates. In particular the coherence overhead is evident for *Chebyshev* and *Norm*, making performance decrease.

The SB-based configuration (*SB*) improves performance for a subset of the benchmarks. For the *LUs*, the SB reduces the execution time of the 25%, while on *Radix* it features more than 40% reduction, with respect to *MEM*. The same behavior is for *Chebyshev*, *15-puzzle* and *Norm*.

SB-based synchronization does not suffer from coherence protocol overhead, resulting in better performance than *COH*: 25% improvement for *LUs*, 40% for *Radix* and 10-30% for *Chebyshev*, *15-puzzle* and *Norm*.

Load balance characteristics are important in determining how much the SB impacts on system behavior. If the load is not balanced, busy-wait synchronization makes some PEs wait. If spinning synchronization operations are software implemented, during these time slots, PEs do a lot of polling, while, using the SB, no polling is required, speeding up the execution. As shown in [22], both *LUs* and *Radix* spend long time in synchronization (35%, 10% of the execution time), featuring significant load imbalance, while *FFT* is the most balanced benchmark (less than 1% of the execution time is synchronization). Due to this reason, for some benchmarks (*FFT* and *Matrix*) the impact of the SB is negligible, while, especially for the *LUs* and *Radix*, which are synchronization-intensive applications, the SB significantly improves performance.

For what concerns *Bio* and *QWST*, the SB has little effect on the performance, while it works in reducing NoC traffic, as observed in Figure 4 for *Bio*. These benchmarks spend a large fraction of the execution time for computation, so the SB effects can't be observed from system performance analysis.

Figure 5(b) shows energy breakdown for each benchmark and synchronization schemes. We consider three components of the total energy: the PE energy component (which includes the whole node energy consumption, including caches); The memories component which refers to shared memory; The NoC component, related to interconnect energy consumption.

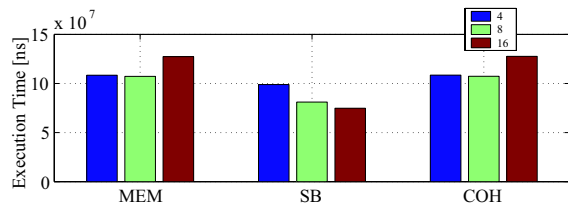Caching synchronization variables (*COH*) is effective in

Figure 6. Execution time scaling trends for the _LU-2_ application

reducing the contention due to polling. In fact, it reduces significantly memory energy, impacting on those benchmarks with large synchronization overhead. _LU-1_, _LU-2_, _Radix_, _Bio_ and _QWST_ feature energy reduction up to 30% with respect to _MEM_ configuration. On the other hand, caching synchronization variables means that coherence protocol has to be enforced to guarantee consistency. _Norm_, _Chebyshev_ and _15-puzzle_ are benchmarks where energy spent to maintain coherence is not compensated by memory energy reduction.

The _SB_ behaves in a similar way, but avoids the overhead of coherence protocol, resulting in 10–30% energy reduction with respect to _COH_ configuration, for the _LUs_, _Radix_, _Chebyshev_, _15-puzzle_ and _Norm_, by reducing the NoC and the PE energy component. In fact, network transactions due to coherence are avoided and performance increase of the SB-based solution impacts on the PE energy.

In Figure 6, scaling trends for the _LU-2_ benchmark are shown. The SB allows effective application scaling when the system size grows (4, 8 and 16 cores). _MEM_ does not scale well since synchronization variables reside in the memory and longer network transactions are needed as system size increases. The _COH_ configuration relies on coherence protocol to avoid the polling, but this features increasing traffic as the system becomes larger. For the SB-based architecture, for each processor which requests to poll a shared variable a constant number of network transactions, independent of the system size, is generated. Scaling results are similar for all the synchronization-intensive benchmarks.

7. Concluding Remarks

In this paper, we suggest shared memory architecture, optimized to manage synchronization operations, by means of a dedicated hardware unit: the Synchronization-operation Buffer (SB). This solutions is low-complexity, avoiding the additional cost of coherence protocol, which is often used by state-of-the-art synchronization solutions. The SB features up to 40% performance improvement and 30% energy reduction with respect to a system based on coherence protocol. The SB is effective especially for those applications featuring significant synchronization overhead. As the system scales up this phenomenon becomes more and more evident.

8. Acknowledgements

References

[1] http://www-flash.stanford.edu/apps/SPLASH/.
[2] STMicroelectronics industrial data.
[3] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O'Neill, J. Othmer, E. Sckinger, K. Singh, J. Sweet, and C. Terman. A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP. _IEEE JSSC_, March 2000.
[4] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 95/7, WRL, 1995.
[5] T. Anderson. The performance of of spin lock alternatives for shared memory multiprocessors. _IEEE Trans. on Parallel and Distributed Systems_, January 1990.
[6] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS macros for shared-memory multiprocessor environments. Technical Report UPC-DAC-1997-07, UPC-DAC, 1997.
[7] A. Beda, N. Gatti, and F. Amigoni. Heart-rate pacing simulation and control via multiagent systems. In _Proc. of Agents Applied in Health Care_, 2004.
[8] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital tv systems. _IEEE Design & Test of Computers_, September/October 2001.
[9] A. Kagi, D. Burger, and J. R. Goodman. Efficient synchronization: let them eat QOLB. In _Proc. of ISCA_, 1997.
[10] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. _IEEE Micro_, pages 21–29, March/April 2005.
[11] M. Loghi and M. Poncino. Exploring energy/performance tradeoffs in shared memory MPSoC: Snoop-based cache coherence vs. software solutions. In _Proc. of DATE_, 2005.
[12] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread Itanium processor. _IEEE Micro_, pages 10–20, March/April 2005.
[13] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. _ACM Trans. on Computer Systems_, 9(1):21–65, 1991.
[14] T. Mudge. Power: A First Class Constraint for Future Architectures. In _HPCA-6_, 2000.
[15] G. Palermo and C. Silvano. PIRATE: A Framework for Power/Performance Exploration of Network-On-Chip Architectures. In _Proc. of PATMOS_, 2004.
[16] D. A. Patterson and J. L. Hennessy. _Computer Architecture: A Quantitative Approach_. Morgan Kaufmann, 2003.
[17] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to high-speed traffic management. In _Proc. of CODES-ISSS_, pages 48–53, 2004.
[18] W. Qin and S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. In _Proc. of DATE_, 2003.
[19] R. Rajwar and J. Goodman. Transactional execution: Toward reliable, high-performance multithreading. _IEEE MICRO_, November/December 2003.
[20] A. Sinha, N. Ickes, and A. Chandrakasan. Instruction level and operating system profiling for energy exposed software. _IEEE Trans. on VLSI_, 11(6), December 2003.
[21] W. Wolf. The future of multiprocessor systems-on-chips. In _Proc. of DAC_, 2004.
[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In _Proc. of ISCA_, 1995.