Cache Coherence Tradeoffs in Shared-Memory MPSoCs

MIRKO LOGHI
Università di Verona
MASSIMO PONCINO
Politecnico di Torino
and
LUCA BENINI
Università di Bologna

Shared memory is a common interprocessor communication paradigm for single-chip multiprocessor platforms. Snoop-based cache coherence is a very successful technique that provides a clean shared-memory programming abstraction in general-purpose chip multiprocessors, but there is no consensus on its usage in resource-constrained multiprocessor systems on chips (MPSoCs) for embedded applications. This work aims at providing a comparative energy and performance analysis of cache-coherence support schemes in MPSoCs. Thanks to the use of a complete multiprocessor simulation platform, which relies on accurate technology-homogeneous power models, we were able to explore different cache-coherent shared-memory communication schemes for a number of cache configurations and workloads.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement techniques

General Terms: Measurement Performance

Additional Key Words and Phrases: Cache coherence, low power, multiprocessor, system-on-chip

1. INTRODUCTION

The rapid advances of silicon technology have made it possible to fabricate hundreds of million transistors on a chip, which can be used to build small- to medium-scale single-chip complete multiprocessors, including memories. Such single-chip multiprocessors come in many shapes and could be broadly classified according to the target application domain and on architectural organization.

A qualitative classification that privileges the architectural dimension is shown in Figure 1, where some existing single-chip multiprocessors are placed

Authors' addresses: Mirko Loghi, Università di Verona, Verona, Italy; email: loghi@sci.univr.it; Massimo Poncino, Politecnico di Torino, Torino, Italy. Luca Benini, Università di Bologna, Bologna, Italy.

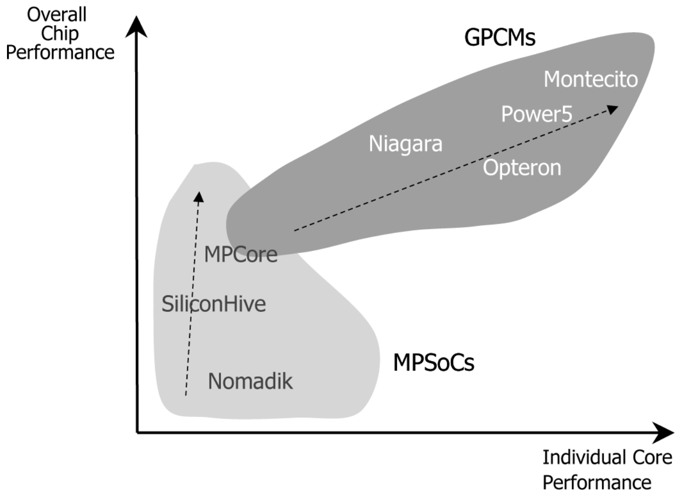


Fig. 1. Classes of single-chip multiprocessors.

in a two-dimensional space defined as follows. The x-axis denotes the performance of the individual computing elements (i.e., the cores), whereas the y-axis denotes the overall chip performance, defined as the cumulative performance of the individual cores.

We notice that, although qualitative, the plot exhibits two rough regions defined by quite well-defined architectural characteristics. A first region, in which overall chip performance is obtained as a result of using few high-performance cores (a small slope of the trend line) and a second one in which overall chip performance (usually much smaller than for architectures belonging to the other region) is obtained as a result of using possibly large numbers of low- to medium-performance cores (a high slope of the trend line).

Interestingly, these areas also correspond to chip multiprocessors with different application targets; in order to distinguish between them, we use specific names for these two categories. We call the former class *general-purpose chip multiprocessors* (GPCMs), which integrate a small number of advanced processor cores (e.g., Itanium 2, UltraSPARC) and large caches in a tightly connected cluster, and that typically target the high-end server market and are often used as building blocks for large-scale supercomputers [McNairy and Bhatia 2005; Geppert 2005; Kalla et al. 2004; Keltcher et al. 2003].

In contrast, we define the other class *multiprocessor systems-on-chip* (MPSoC), which contain simpler cores and many application-specific heterogeneous coprocessors, embedded memories and peripherals, and are targeted for embedded applications (multimedia, video, graphics) in tightly cost- and power-constrained markets (e.g., smart phones, home entertainment centers) [ARM11 MPCore ; SiliconHive; Cumming 2003; Richardson 2002; Ackland et al. 2000; Philips Semiconductor; Grammatikakis et al. 2003].

The focus of this paper is the study of power–performance tradeoffs in supporting shared-memory programming models in MPSoCs. Needless to say, power consumption is also a concern in GPCMs. In the case of GPCMs, however, fully hardware-supported cache coherence is an undisputed requirement, because it makes much easier to support general purpose application workloads. Recent papers by Ekman et al. [2002a, 2002b] have studied the power consumption of snoop-based cache coherence using a full-system simulation approach targeted to GCMPs.

The picture is much less clear in MPSoCs; although some MPSoCs explicitly support cache coherence in HW [e.g., ARM11 MPCore ; Ackland et al. 2000], other devices on the market rely on noncache-coherent architectures [e.g., Cumming 2003]. This is motivated by the fact that embedded applications are usually carefully tuned to the target hardware platform and interprocessor communication is often performed by explicitly managing shared-memory areas, without much hardware support for a full shared-memory abstraction. Clearly, the ease of programming in a fully cache coherent memory space could simplify embedded application development, but designers would reluctantly accept this approach if this should affect the energy efficiency of the architecture.

Our work sheds some light on this open issue. We set up a complete and accurate environment for exploring the energy efficiency of cache coherence in a MPSoC context, using cycle-accurate simulation and power models that are technology-homogeneous (i.e., all obtained from characterization in the same $13\mu m$ technology). We compared three alternative approaches to cachecoherence: in the first one (*hardware-based*), cache coherence is imposed by a specific device implementing a snoopy protocol; in the second one (*noncoherent*), coherence is enforced by preventing the caching of shared data. In the third scheme (*OS-based*), the burden of coherence is left to the operating systems IPC primitives (that are, in our platform, based on message passing).

Our analysis allows us to derive some interesting and nontrivial conclusions. First, results show that an OS-based coherence solution is extremely inefficient both power- and performance-wise (up to 7x), independently of the benchmark and of the cache configuration. Second, we show that cache coherence is not always convenient in terms of either performance and energy; this strongly depends on hardware features, such as cache size, as well as on the characteristics of the application such as the access patterns for shared variables. The latter, in particular, allows to define some high-level guidelines for writing embedded software for cache-coherent MPSoCs.

This paper is organized as follows. Section 2 surveys previous works on cache coherence in GCMPs and MPSoCs. Section 3 provides a description of the simulation platform used in this work, its implementation, as well as its software architecture. Section 4 describes the coherence schemes used in our exploration and section 5 describes and analyzes simulation results. Finally, section 6 summarizes the work and draws a few conclusions.

2. BACKGROUND AND PREVIOUS WORK

The problem of cache coherence has been thoroughly studied by researchers and a vast literature on the subject is available. Broadly speaking, approaches for solving the cache-coherence problem in multiprocessor systems fall into two major classes: hardware-based and software-based approaches. The former impose cache coherence by adding suitable hardware which guarantees coherence of cached data, whereas, the latter impose coherence by limiting the caching of some shared data to when it is safe to do it; this can be done by the programmer, the compiler, or the operating system. For a survey of hardware-based cache-coherence solutions the reader is referred to Stenström [1990], Tomasevic and Milutinovic [1994], and Culler et al. [1997]; software-based cache-coherence solutions are reviewed in Tartalja and Milutinovic [1997].

Hardware-based cache coherence solutions hereafter called cache-coherence *protocols* can be further classified according to two orthogonal dimensions [Stenström 1990]:

1. *The type of interconnect of the multiprocessor architecture*. When processors are connected through a shared medium (such as a bus), protocols can use broadcasting to enforce coherence. These protocols are called *snoopy protocols*. These schemes apply to small-scale bus-based multiprocessors, because of the limited scalability of buses. In absence of a shared medium as interconnect (e.g., if a crossbar connection is used), snoopy protocols are replaced by *directory-based* protocols (which are outside the scope of this paper).
2. *The type of cache-coherence policy*. There are essentially two options: a *write-invalidate* and a *write-update* policy. In the former scheme, whenever any cache line L is written, the coherence protocol invalidates all copies of L in other caches; in the latter one, the protocol updates those lines with the new value, which is being written.

   Invalidation-based solutions are more common in coherence protocols because they are easier to implement in hardware; they are also more efficient than update-based one for large-cache line sizes, since updates require multiple bus transfers. Update-based protocols become more efficient when accessing heavily contended lines, since subsequent accesses to those lines will result in a cache hit, thanks to the update.

Multiprocessor architectures (and, hence, cache coherence) have been historically designed with performance in mind. Therefore, the impact of coherence schemes on energy has not been considered until the tight energy constraints of single-chip multiprocessors made the problem relevant. Early work on this topic include the Jetty scheme [Moshovos et al. 2001], where a small structure (Jetty) is attached to each cache so as to filter out useless snoop accesses. Instead of doing a tag-lookup directly, the Jetty is checked first; if no copies of data exist, a cache access is avoided, thus achieving significant energy savings. Another approach is *serial snooping* [Saldanha and Lipasti 2003], based on the assumption that if a miss occurs in one cache, it is possible to find the block in another cache without having to check all the other caches. Both schemes have been devised for multichip SMPs; they have been evaluated on GPCMs in Ekman et al. [2002a], where it was shown that these solutions are quite ineffective. The same authors have also proposed an energy-efficient cache-coherence scheme for virtual caches in GPCMs [Ekman et al. 2002b].

None of these approaches is explicitly meant for MPSoCs, for which much tighter energy and area constraints do exist. As a matter of fact, all these

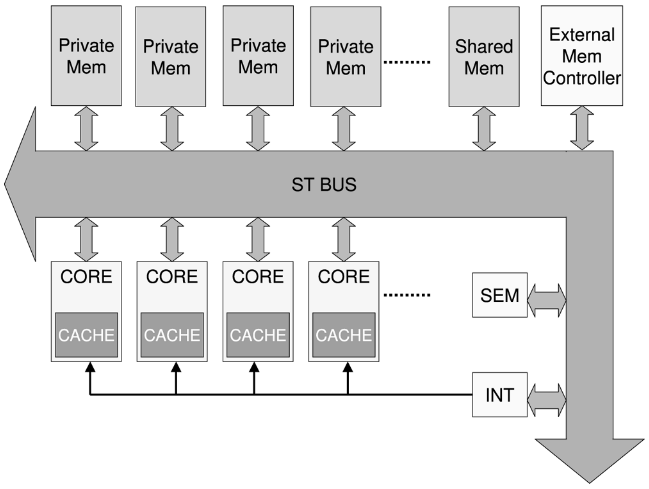


Fig. 2. Hardware architecture.

schemes require quite high computational and/or hardware overhead and assume nonrealistic software architectures. For example, in Ekman et al. [2002a, 2002b], no operating system is assumed to be executing in the system.

In this work, we propose the first energy/performance analysis in MPSoCs, that includes an embedded operating system. The result is a comparison of basic hardware-based coherence schemes, with minimal impact on the architecture, with respect to schemes imposed at the software level.

3. THE MULTIPROCESSOR PLATFORM

Figure 2 shows the architectural template of the multiprocessor simulation platform used in this work, called MPARM [Loghi et al. 2004]. It consists of (i) a configurable number of 32-bit ARM processors, (ii) their private memories, (iii) a shared-memory, (iv) a hardware interrupt module, (v) a hardware semaphore module, and (vi) the interconnect.

The processor cores are modeled by means of an adapted version of a GPL-licensed instruction-set-simulator (ISS) called SWARM [Software ARM], written in C++ and embedded into a SystemC wrapper. Each ISS contains his own cache, possibly split in instruction and data cache. Memories and all other devices are implemented in SystemC in a straightforward fashion. The semaphore module and the interrupt device are used to handle the synchronization among the cores. The former provides a *test-and-set* operation to the software, while the latter allows a processor to send an interrupt request to another core. Both these specialized hardware devices are mapped into the core's address space. To send an interrupt to a processor, an application writes a word into a specific location; to do busy waiting on a semaphore, it suffices to read from the proper address and loop until the operation return the correct value (namely, 0). The platform is entirely described in SystemC, accurate at the signal level, except the ISS which performs cycle-accurate simulations of the cores; the resulting simulations are signal- and cycle-accurate, respectively.

The platform is configurable and it allows to specify several parameters:

- *The type of interconnect*. The 32-bit interconnection system can be an *AMBA AHB bus* [AMBA Home Page] or a *ST-Bus* (a proprietary bus by STMicroelectronics). In addition, the topology of the ST-bus can be a *shared bus* configuration, a *full crossbar* configuration, or an intermediate *partial crossbar* topology. In this work, we focus on an ST-bus shared bus configuration.
- *The number of processing elements*.
- *The cache parameters*. In particular, we can specify the organization (split or unified), the type (direct-mapped, fully-associative or set-associative), the size, and the line size.
- *The memory parameters*. In particular, we can specify the size and the latency. This also applies to the dedicated hardware (which is viewed as a memory) and the size of the memories.

3.1 External Memory

At the board level, MPSoC platforms always interact with external memories such as DRAMs and NV-RAMs (FLASH, EPROM, etc.). However, access to these memories is generally performed through explicit data transfers using dedicated memory controllers [Carter et al. 1999; Hong et al. 1999; Gries 2000]. This is in sharp contrast with large-scale general-purpose multiprocessors, where the complete memory hierarchy is managed in a uniform way, though virtual memory abstraction.

For this reason, in our analysis, we focus on on-chip memory. We assume that off-chip memory accesses are explicitly managed through DMA-based external memory controllers: management of off-chip memory in embedded applications is outside the scope of this work (the interested reader is referred to the relevant literature [Gries 2000; Marchal et al. 2003]). In the following sections, we assume that the working set for the applications is transferred on-chip using background DMA transfers and that the data is always available on-chip when needed by the application. The interaction between off-chip and on-chip memory traffic is a topic for future research.

3.2 Software Implementation

Figure 3 shows the software architecture of MPARM. Names in parentheses inside the boxes denote the names of the corresponding C++ class.

The ISS is modeled as a C++ class (*CArmProc*) and is embedded into a SystemC wrapper (*armsystem*). The wrapper allows the usage of the ISS in a SystemC environ, while the bus interface (implemented in the class *STBus_initiator*) is in charge to translate the ARM requests toward the bus, to bus-specific protocol requests.

In addition, the core contains its cache as a data member of type *CCache*. This is a base class from which other classes that implement specific cache types can be implemented: the fully associative (*CAssociativeCache*), the direct-mapped (*CDirectCache*), and set-associative cache (*CSetAssociativeCache*). The

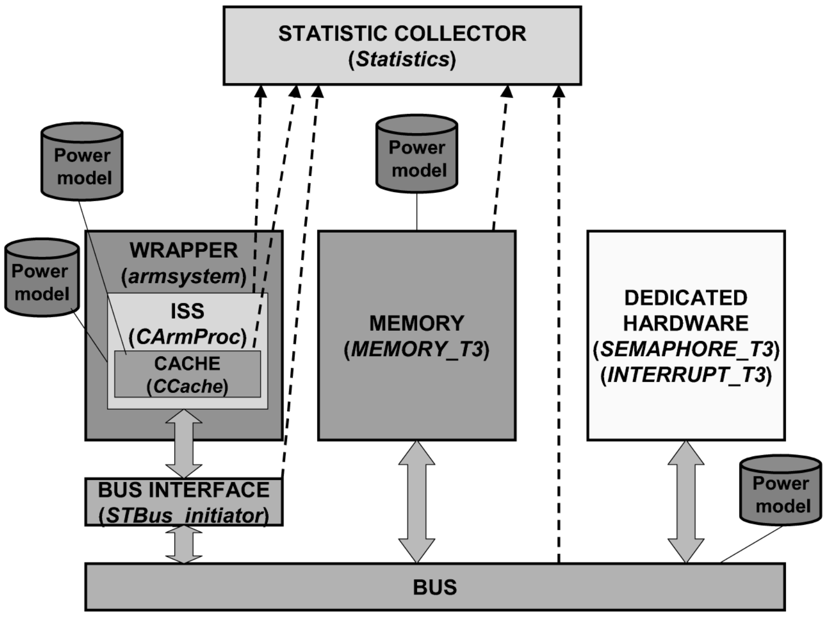


Fig. 3. Software implementation of the platform.

base class provides a common interface to access to the data from the code executed by the ISS.

The implementation of memories and of dedicated hardware is straightforward, because of their simple functional behavior. Memories are simply a class (*Memory_T3*) which embeds an array of data; the only potential difficulty lies in the interface toward the bus. The memory, in fact, must be able to understand the bus protocol and communicate accordingly. Dedicated hardware modules (*Semaphore_T3* or *Interrupt_T3*) can be thought of as a variant of memories. They also receive read or write requests and must act accordingly, depending on their specific operations. The overall platform is instantiated in a dynamical way at the very beginning of the simulation in the `sc_main` method.

Besides the classes, which implements hardware devices, we developed some other class to perform data collection for performance and power consumption. The main one is the *Statistics* class, which contains several methods to handle and record events. For instance, the `inspectMemoryAccess` and the `InspectCacheAccess` are called when a memory or a cache module is activated, respectively.

3.3 Power Models

Concerning power analysis, the platform provides accurate power models associated with each component. All models are cycle-accurate and have been characterized on a 0.13-μm technology by STMicroelectronics and validated on silicon implementations of the various components. Using the models, the platform can provide the energy spent by any of the different components on a cycle-by-cycle basis. The models are functions, which compute the energy spent by the correspondent device, using information on the device's internal state.

For the cores we have used a instruction-based power model (see [Benini et al. 2001] for details). We distinguish between a *RUNNING* and an *IDLE* state, with distinct values of power consumption. An ARM core can be in *IDLE* state when its internal pipeline is stalled, for data or instruction dependency, or when it is waiting for some data from the bus. In both cases, the core is consuming a smaller (yet nonzero) amount of energy with respect to the *RUNNING* state. Power consumption figures have been obtained from an implementation of an ARM7 on a 0.13-μm technology by STMicroelectronics (0.055 mW/MHz for the *RUNNING* state, and 0.036 mW/MHz for the *IDLE* state).

For the memories (both caches and private memories), we have used an analytical model [see Chinosi et al. 1998], derived from interpolation of data extracted from a memory generator by STMicroelectronics for the same 0.13-μm technology. The model is parameterized with respect to the memory size (in 32-bit words), and has been derived by least-mean square regression of a set of energy values obtained with the memory generator for different memory sizes. We explicitly distinguish between read and write accesses (for which there are two different power models).

The caches are regarded as a special type of memories consisting of two distinct cell arrays, the data and the tag memory. Given the fixed size of the memory word (32 bits), the cache parameters automatically define the size of the tag array(s) and the size of the data array(s). For instance, a 4 KB unified, two-way associative cache with 16-bytes lines will have two 24-bit tag arrays and two 128-bit arrays.

To accurately model cache power, cache accesses are decomposed into different access subtypes. For instance, writing a word in a cache consists of (i) a tag memory access and (ii) a data memory access. In addition, if the cache is n-way associative, only the bitlines corresponding to the desired way are activated. This is different, for instance, from the case of a cache refill, in which an entire line is written into the cache.

For this reason, we have defined a richer set of cache access modes, with different power models: *READDATA* and *WRITELINE* when a whole line of the data memory is read or written, *READTAG* and *WRITETAG* when the tag field is read or written and *WRITEWORD* when a single word into a data memory line is written. Therefore, the power spent by a cache operation depends on the operation type, on the cache type (fully associative, direct-mapped, or set-associative), and on the line size, for the last parameter affects the size of the tag-memory.

The power model for the ST-bus is relative to the same 0.13-μm technology and has been provided by STMicroelectronics. It computes the power spent during a clock cycle using the number of *cells*, which are in transit on the bus. This datum is available thanks to the signal accuracy of the simulation, which allows to know how many devices are transmitting on the bus by simply analyzing the request and grant signals.

The mechanism used to invoke the various power models is described in Figure 4.

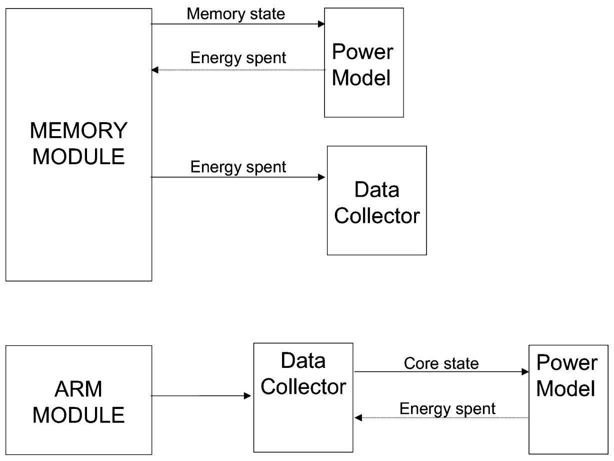


Fig. 4. Invocation of power models.

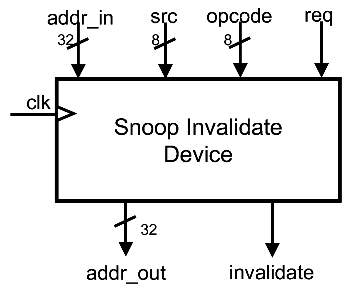
When a given module is activated, the related power module function is invoked with the actual parameters carrying information on the device state. For the ST-bus power model, this data is the number of cell in transit on the bus, while for memories and caches it is the memory size and the access type. The power module function returns to the caller (the module implementation) the amount of energy spent for the current operation. The module moves this value to the data collection routines, which are in charge to gather and record information about performance and energy of the system. For the ARM core, the information flow is different. The ISS, in fact, does not run when the core is stalled waiting for a bus response. Since the core is consuming energy also when idle, to collect this energy the power model is invoked from the data collector routine, which is activated at each cycle, and keeps track of the state of the core.

3.4 Cache-Coherence Support

The base MPARM architecture does not support cache-coherence, in the sense that no hardware support is used to enforce it. We have enhanced the platform by adding hardware coherence support based on various policies, categorized according to two dimensions: *cache-write policy* (write-through vs. write-back) and *cache-coherence policy* (write-invalidate vs. write-update).

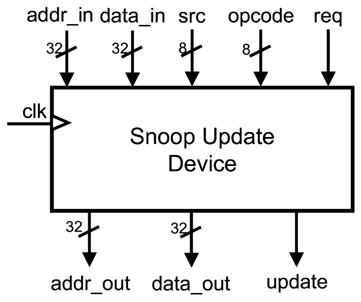
For the WT write policy we have implemented two simple protocols: write-through invalidate (*WTI*), and write-through update (*WTU*). Using these protocols, whenever a processor writes a shared variable into its cache, it also performs the write into the RAM. The cores that have that variable into their caches then perform the appropriate operation (invalidation for WTI, or update for WTU).

Concerning the WB policy, we have implemented the well-known MESI protocol and the Berkeley (BER) protocol. These protocols assume four possible states for a cache line: *Modified*, *Exclusive*, *Shared*, and *Invalid* for the MESI, and *Invalid*, *Valid*, *Shared-dirty*, and *Dirty* for the BER. Cache tags must thus



```
SnoopDeviceInvalidate(addr_in,src,opcode,req)
{
if ((req==1) &&      /* a request */
    (opc[3:0]==2) && /* a write */
    (src != 0) &&    /* by another core */
    ((address>=LOW) && (address<HIGH))) {
          invalidate = 1;
          addr_out = addr_in;
} else {
          invalidate = 0;
          address_out = 0;
}
}
```

(a)



```
SnoopDeviceUpdate(addr_in,data_in,src,opcode,req)
{
if ((req==1) &&      /* a request */
    (opc[3:0]==2) && /* a write */
    (src != 0) &&    /* by another core */
    ((address>=LOW) && (address<HIGH))) {
          update = 1;
          addr_out = addr_in;
          data_out = data_in;
} else {

          update = 0;
          address_out = 0;
          data_out = 0;
}
}
```

(b)

Fig. 5. Operations of the WT snoop device for the invalidate (a) and update (b) policies.

be enlarged with two additional bits with respect to the baseline (without cache coherence) cache architecture.

Besides multiple states of a line, WB-based protocols are more complicated from the control point of view. First, they require the inspection of the RAM reads, besides writes. In fact, each core must know if a shared variable contained in its cache is also present in some other cache. If a processor wants to write a variable owned by other cores, it must signal this fact to them. Second, if a core modifies a variable into its cache and it does not update the RAM, since the policy is WB, it must provide the new value of such variable in response to future requests of other processors.

3.4.1 *Hardware Implementation.* The hardware snoop devices are in charge to enforce the cache coherence protocols. The snoop devices sample the bus signals to detect the transaction on the bus, its relevant data and the core involved.

Figure 5 shows the interface and the basic behavior of the snoop devices developed for WT policies. For WT-based policies the significative operations are only the write ones: when a write request on the shared memory is coming from another processor (detected from the observation of the corresponding bus signals), the corresponding action is performed, i.e., invalidation for the WTI policy, rewriting of the data (update) for the WTU one. Write operations are performed in two steps. The first one is performed by the core, which asserts the signals on the bus, while the second one is performed by the target memory, which sends its acknowledge. The write ends when the second step is complete

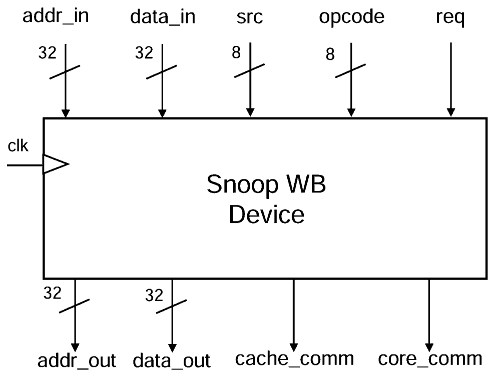


Fig. 6. The WB snoop device.

and when it is the right time, for the snoop device, to interact with the cache. Of course, the device must ignore writes issued by its core. Notice that the update device is slightly more complex than the invalidate one. It features extra I/O interface for the data, since in this case we need to write a value to the updated cache location.

For WB-based policies the snoop device's behavior is more complex (Figure 6 shows the device's interface). The crucial operations to detect are both the reads and the writes. Furthermore, the snoop device must interact as with the cache as well with the processor core. Due to the cache coherence protocols for the WB policy, some read operations may be aborted, because the data in memory are not updated. The updated data are actually owned by another core, and reside in its cache. In such a situation the snoop device must signal to the core that issued the read that the read operation must be interrupted. Next, the snoop device must also force the owning core to put the data on the bus, actually performing a write in memory. Then the I/O interface of the snoop device for the WB case, holds an output port to carry the commands to the core.

Of course, even in the WB case, all the operations are performed in two steps. A write operation consists of a request and of an acknowledge, while a read operation consists of a request and of a response which carries the data. The snoop devices usually interact with the caches and with the cores at the end of the second phase, but sometimes it can operate at the end of the first phase (as, for example, when a read must be interrupted).

In our processor node, synchronization between the core and the snoop device is handled using a mutual exclusion register, which allows the core, or alternatively, the snoop device, to lock access to the cache memory. Hence, we can model accurately the performance penalty of contention for cache access.

3.5 The Operating System

A port of a real-time operative system called RTEMS (real-time executive for multiprocessor systems [RTEMS Home Page]) has been realized for this platform. User applications do link the RTEMS libraries to gain access to the functionality of the embedded system, in the form of a set of system calls.

RTEMS is a lightweight and flexible OS for embedded systems, it offering good support for multiprocessing and providing native calls for communication and synchronization in such multiprocessor environments. Its targets are homogeneous and heterogeneous multiprocessing systems, for which it provides multitasking capabilities, flexible scheduling, and a high degree of user configurability.

Interprocess and interthread communication in RTEMS rely on *message queues*. A thread, which need to communicate with another thread, must open a *message queue*, which can be its own (local queue) or owned by another thread (remote queue). Both tasks involved in the communication must open the same queue, identified by a global ID and one of them will own the queue. The transmitter thread then uses the `rtems_message_queue_send` to put data into the queue, while the other uses `rtems_message_queue_receive` to retrieve them. The behavior of these two system primitives depends on the owner of the queue. It is different, in fact, to use a `rtems_message_queue_send` or a `rtems_message_queue_receive` on a remote queue with respect to a local queue. However, the differences in the queue handling are managed from RTEMS and are totally transparent to the user.

4. ENERGY IMPACT OF CACHE COHERENCE

To assess the energy/performance efficiency of cache-coherence in MPSoCs, we compared three different schemes for enforcing cache-coherence, each one corresponding to a different *programming model*, i.e., the way coherence is made available to the programmer. It is worth emphasizing that all the three models assume *strict memory consistency* [Culler et al. 1997].

Under this consistency model, any read to a memory location X returns the value stored by the most recent write operation to X. This is equivalent to the semantics of uniprocessors and it implicitly assumes the existence of absolute global time so that the notion of "most recent" is not ambiguous.

The choice of strict consistency is dictated by the fact that is simple; although more relaxed consistency models are known to provide substantial performance improvements, for MPSoCs simplicity is more important and strict consistency is typically the most appropriate choice [Hill 1998].

The following subsections describe these three schemes. For each scheme, its corresponding programming model will be illustrated through a simple example, namely, a one-item producer–consumer application.

4.1 Hardware-Based Coherence

Under this model, cache-coherence is imposed by the snoop devices attached to the cores and their caches. This scheme implies a program semantics similar to that of an uniprocessor context; shared data can be cached because caches are guaranteed to be coherent. However, the programmer must explicitly deal with synchronization (e.g., for mutual exclusion) of shared data.

Figure 7 shows a typical textbook pseudocode for our working example. The use of the `shared` variable `value` is guaranteed to be coherent. However, the

```
shared int value;
shared sem mutex = 1, filledslots = 0, emptyslots = 1;

void producer(void)
{
    while (TRUE) {
        Wait (emptyslots);
        Wait (mutex);
        value = Produce ();
        Signal (mutex);
        Signal (filledslots);
    }
}

void consumer (void)
{
    while (TRUE) {
        Wait (filledslots);
        Wait (mutex);
        Consume (value);
        Signal (mutex);
        Signal (emptyslots);
    }
}
```

Fig. 7. Producer-consumer pseudocode with one-item buffer.

programmer must use a synchronization variable (a semaphore, in the example) to regulate access to the data.

4.2 Software-Based Coherence

The second coherence scheme is straightforward: *shared data are not cached*. Although this solution may appear trivial, it is the basis of typical software-based schemes. More advanced schemes belonging to this class require compiler support to perform accurate analysis, which allows caching of some shared data when it is safe to do it [Tartalja and Milutinovic 1997]. In our example, the code differs only marginally from the one in Figure 7, namely, the `shared` keyword used on a variable automatically implies the fact that it cannot be cached. It will be allocated in a noncacheable memory segment at program loading time.

4.3 OS-Based Coherence

The third scheme leaves to the OS the task of guaranteeing coherence, through its IPC abstraction offered by its API. Specifically, RTEMS offers a communication infrastructure based on *message queues* [RTEMS Home Page], that are shared objects implemented as a pool of buffers, called *packets*. In order to communicate with each other, remote processes obtain packet buffers using the the global identifier of the queue. Synchronized accesses to these buffers are realized by means of *locks*, which can be thought of as an equivalent of a hardware *test-and-set* primitive, implemented by polling a given location of the shared-memory.

From the programming model point of view, not just coherence, but even the notion of shared data is hidden by the OS primitives. The producer–consumer example becomes then as shown in Figure 8, where `send` and `receive` denote the generic primitives for communication between remote processes. In the example, communication is established by explicitly specifying the peer process involved in the communication.

The two versions of software cache-coherence for WT and WB will be denoted by WTS (software write-through) and WBS (software write-back), for the sake of conciseness.

```
void producer(void)
{
    message m;
    int item;
    while (TRUE) {
        item = Produce()
        build_message(&m, item);
        send(consumer, &m);
    }
}
```

```
void consumer (void)
{
    int item;
    message m;
    while (TRUE) {
        receive(producer, &m);
        item=extract_item(&m);
        consume_item(item);
    }
}
```

Fig. 8. Producer–consumer pseudocode with one-element buffer.

5. EXPERIMENTAL RESULTS

5.1 Benchmark Description

In order to accurately compare the schemes described in the previous section, we have chosen a set of parallel programs, which exhibit different access patterns to the shared-memory. The synchronization among processes relies on OS primitives, for the applications that use the OS and on explicit manipulation of the hardware semaphore module for the others. In particular, the synchronization is always based on the *test-and-set* hardware feature. Addresses corresponding to semaphores are never cached, because it is not possible to perform a *test-and-set* access on a cache memory. The benchmarks are split into two sets. The first set includes two synthetic parametric benchmarks:

1. *A producer–consumer application* (`PCx,y,z`). The application is parameterized with respect to the number of producers `x`, consumers `y`, and the size of the FIFO queue `z`. All the producers write their data in a queue and all the consumers read from the same queue. When the queue is full, the producers busy-stall polling the semaphore, while consumers busy-stall when the queue is empty. It is not relevant which consumer gets the data written by a producer, so the only synchronization point is related to the queue access. Each process performs a fixed number (`N`, set at 1000 in our experiments) of operations (reads or writes to the queue).
2. *An application implementing the readers–writers problem* (`RWx,y,z,w`). The meaning of `x`, `y`, and `z` is the same as for the `PC` application, while `w` denotes the relative speed of the readers with respect to the writers. The writers and the readers use the same shared object for data exchange. Here the aim of synchronization is to avoid multiple, simultaneous write accesses, while simultaneous reads must be allowed. Furthermore, no reads are allowed during a write and each writer must have exclusive access to the shared data. Unlike the previous case, multiple writes before one read and multiple reads before one write are possible In this case, each process (reader or writer) also accesses the same number of times `N` the shared object. Varying the relative speed of the processes will change the order of the memory accesses, but not their total number.

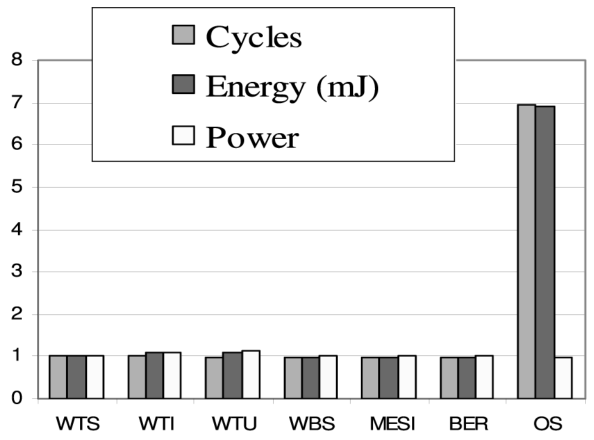


Fig. 9. Relative energy and performance for the producer–consumer application.

The second set of benchmarks consists of a set of small kernels implementing well-defined functionalities:

1. A parallel matrix multiplication (*MM*). Each processor uses the entire source matrices and produces a slice of the result matrix. This program is written so as to maximize the sharing of the read-only variables (the source matrices) and to minimize the sharing of the variables that are written.
2. A parallel *FFT*.
3. A parallel LU matrix decomposition (*LU*).

The last two application are taken from the *SPLASH-2* benchmark suite [Singh et al. 1992].

5.2 Analysis

We have compared the cache-coherency schemes using power P, execution time T (in cycles), energy $E = PT$, and the energy-delay product $EDP = ET = PT^2$. To allow an uniform comparison, all results are normalized with respect to the case of SW cache-coherence and WT policy.

As a first experiment, we compared the various coherence schemes on the producer–consumer application (`PC2,2,16`). Figure 9 shows performance, energy, and power results of the various coherence schemes. For WT policy, *WTI* (*WTU*) denotes hardware-based coherence using invalidate (update) protocol and *WTS* denotes software-based coherence. For WB policy, WBS denotes software-based coherence, while MESI and BER denote the corresponding protocol. Last, *OS* denotes coherence imposed by OS communication primitives.

The most striking results emerging from this comparison is the intrinsic inefficiency of OS-based coherence. The results show that in MPSoCs the price paid in terms of performance and energy for an OS-based user-friendly programming model is very high. The existence of some overhead because of the OS is generally to be expected, but, in this case, it appears to be quite significant.

There are two main sources of this OS overhead. The first one is of general nature, and has to do with to the abstraction that the OS provides to the programmer; since the same code must be used for various operating conditions (architectures and applications), more instructions are required to perform a message exchange. In other terms, abstraction implies generality and, as such, it cannot exploit the knowledge of underlying hardware.

The second source of overhead is because of the flow of data across the communication infrastructure. The OS abstracts the notion of shared data and it provides, in our case, a message-passing environment to the programmer. In such a paradigm, all shared data are actually exchanged from a core to another one through the interconnect. Conversely, in HW- and SW-based coherence schemes, only data related to the shared variables that are really modified have to be transferred, generating traffic on the bus. Although the overhead because of the OS tends to decrease for increasing granularity of the exchanged data (unlike the case of non-OS-based communication), data granularity is not always an independent variable for the software designer, who has to take into account the characteristics of the program and their constraints (e.g., as in multimedia application).

As a conclusion, the choice of the OS clearly impacts the results of Figure 9. However, RTEMS implements a communication infrastructure whose quality is comparable to real-life OSs for MPSoCs. Therefore, even with some architecture-specific optimization of the OS message-passing facility can improve the results, it is unlikely that the 7x overhead can be decreased to a level that makes OS scheme competitive with the other solutions. Therefore, results relative to OS-based coherence will not be considered further in our analysis.

5.2.1 *Synthetic Benchmarks.* Results for the producer–consumer benchmark are shown in Figure 10, that reports performance, power, energy, and EDP figures from top to bottom, left to right. Each plot shows six bars. The leftmost three are relative to WT policies (WTS, WTI, WTU, from left to right), whereas the rightmost three are relative to WB policies (BER, MESI, WBS, from right to left).

The experiments show limited sensitivity to the parameters (# of producers–consumers and buffer size); this is due to the nature of the application; in fact, the average speed for each process is forced to be the same. If the consumer is faster than the producer, the buffer will tend to empty and the consumer will wait for the producer. Conversely, if the producer is faster, the buffer will tend to fill up and the producer will have to wait for the consumer.

In case of speed mismatches, there are substantial differences on the access patterns only for the hardware lock and, because of the non-cacheability of its address space, these variations have the same impact on system performance and power consumption for the cache-coherent solutions, as well as for the noncoherent one.

Concerning the effectiveness of CC policies, we observed that for the WT policies, the WTU solution performs slightly faster than the SW one, but it also has larger energy consumption. The WTI protocol shows higher energy consumption than the other two and has an execution time comparable to WTS, but always worse than the WTU one.

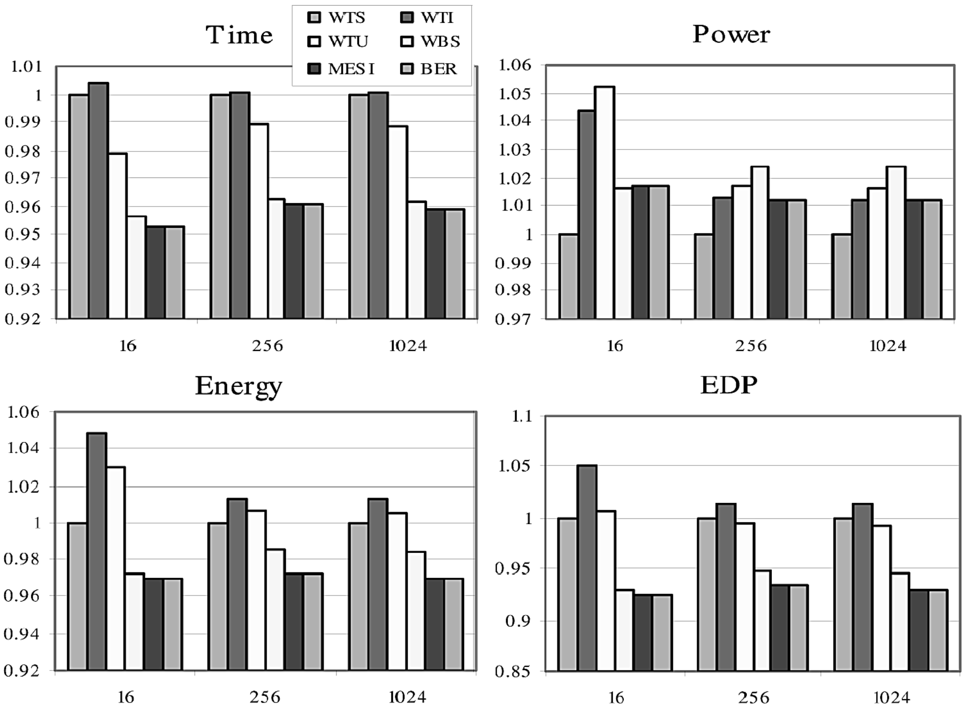


Fig. 10. Relative performance, power, energy and EDP for the PC application for various queue lengths.

The WB policy shows consistently better results with respect to the WT policy. The cache policy, in fact, impacts not only the shared data, but also the private variables. Moreover, the impact on the private data can be more relevant, because we never need to write a dirty cache line in memory only because another core has to update its cache. Thus, using the WB policy on private variables reduces the traffic on the bus and the number of accesses in RAM. There is, as a drawback, the need of a larger tag memory in the cache, to keep track of the status of each cache line (*clean* or *dirty*) and, then, each cache access will have a higher energy cost. The need for a larger tag is even stronger when a cache-coherent protocol is used. In such a case, a cache line can assume one of four states (*modified*, *exclusive*, *shared*, or *invalid* in the MESI protocol). Hence, two additional bits are needed into the tag, and this can reduce the effectiveness of these protocols.

For the PC application, both performance improvements and energy savings are limited and they are more related to the cache policy than to the coherence scheme. The three cases (WBS, MESI, and BER) shows, in fact, very similar results, as for the execution time, as well for the energy dissipation. More generally, all six policies show marginal differences in all metrics: the best improvements are for EDP, that are in the order of 7%.

The second set of experiments refers to the `RW` application, relative to the case of three readers and one writer (similar results have been obtained for other numbers of readers and writers).

We have run three sets of simulations corresponding to the variation of three different parameters: two relative to the application (buffer size and relative speed) and the other relative to the platform (cache size). This benchmark is more interesting than the producers–consumer; here, changing the application parameters (and, in particular, the speed ratio) significantly changes the access pattern to the shared-memory and thus the behavior of the coherence scheme involved.

In simple terms, hardware cache-coherence makes it possible to cache the shared variables. Consequently, the power consumption breakdown shows an increased impact of cache power with respect to bus and shared-memory. Therefore, there are access sequences that benefit from this fact, but also sequences that are penalized.

For instance, consider a WTU scheme and a sequence of consecutive writes in a shared-memory location; each write will cause a sequence of update command on some data cache. The only useful update command, however, is the one which occurs just before a read access on the cache, because the values carried by the other ones are lost. For this access pattern, the WTU scheme is thus quite inefficient.

Conversely, in a sequence of accesses where each write is followed by a read, WTU will be effective, since the update command issued during the write access will load the cache with the data and the following read access will not need a memory and bus access. Note that a useless update command is negative for both power and performance; in fact, since the core cannot access the cache when it is used by the snoop device, some cycles are wasted.

Figure 11 shows performance, power, energy, and EDP, for the RW application varying the speed ratios between the readers and the writers.

The speed ratio directly impacts the access patterns on shared-memory. When the readers and the writers run at the same speed, the cache-coherent schemes present significant advantages (about 20% performance and energy improvement, and 30–35% EDP reduction); in this situation, read and write accesses are, in fact, always interlaced and the accesses to the bus and to RAM are substantially reduced. When readers are slower than writers (speed ratio equal to 0.1), there are many consecutive writes on the shared buffer between two read accesses; this behavior penalizes the cache-coherent architecture. The same holds when readers are faster than writers (speed ratio equal to 10); the readers end their work very early, hence the last portion of the buffer access is composed of only writes because of the writers, which are still running. Still, in both cases, some improvement is observable.

As for the PC application, for the RW the WB policy shows its benefits. The WBS policy usually overcomes the gain because of cache-coherence protocols applied to the WT policy and there is only a case where the WTU policy is slightly faster than the WBS one. With the WB policy, the cache coherence also does not seem to provide remarkable advantages with respect to the SWB case.

In this application the resulting access patterns to the shared object involved in the communication are favorable to the HW-based schemes. The readers and the writers run at the same speed, so the reads and the writes tend to be interleaved. A cache-coherent platform can be more efficient, since it allows the

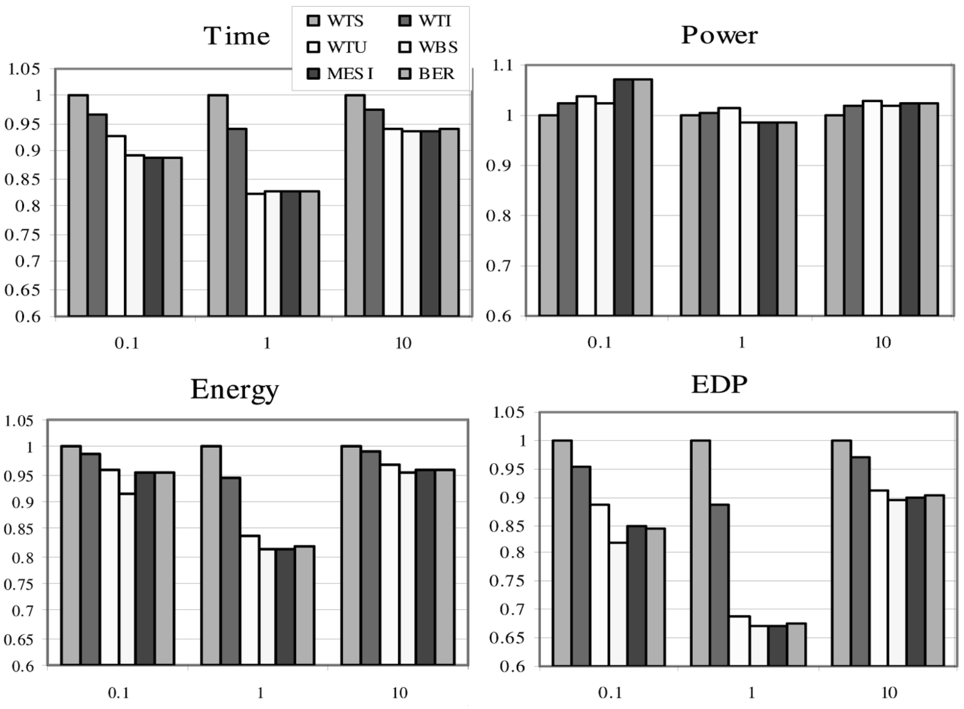


Fig. 11. Relative performance, power, energy, and EDP for the RW application for varying execution speed ratios.

caching of shared data, as the experiments show. By increasing the cache size, we can observe an improvement in performance for both the HW solutions, until the gain saturates. The saturation is because of the point after which the cache is larger than the entire working set (and further increase of the cache size do not provide any benefit). Furthermore, in this situation, the WTU solution is more efficient than WTI, because the invalidation of a cache line will force a read from the shared-memory.

While enlarging the cache size is always beneficial for performance, it negatively impacts energy. In larger caches, the energy required for a single access is larger. Therefore, we can observe an optimal size corresponding to the point where the hit ratio compensates the energy access cost. Moreover, notice that the SW solution is very competitive from an energy point of view, because it reduces the number of accesses to the high-performance, power-hungry data cache.

The analysis of the sensitivity to the size of the object involved in the communication (i.e., the buffer) show a similar trend as for the cache size. In fact, the relation between the object used and the cache size determines the cache hit ratio and the performance. Clearly, using bigger objects causes an increase of the execution time, but this increase is the same for all the cache-coherence schemes adopted. Still, using small objects for the data interchange moves the load of the application from the communication to

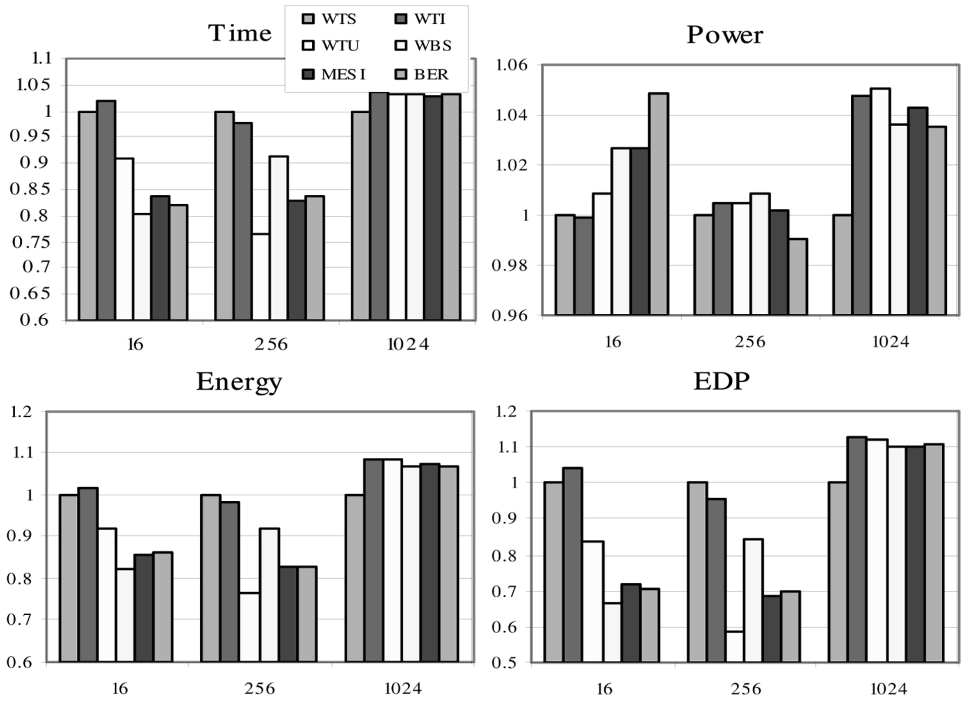


Fig. 12. Relative energy, performance, power, and EDP for the RW application for varying buffer sizes.

the synchronization and the usefulness of the cache-coherent scheme tends to decrease. This is shown in Figure 12, where a shared object of 256 bytes appears to be the best choice, with respect to the 16-byte buffer (too small, synchronization issues became dominant) and with respect to the 1 KB bytes case, as well (the object is too large for the cache used).

5.2.2 *Application Kernels.* The analysis of the synthetic applications shows that *caching a variable is an energy-efficient solution when more reads than writes occurs on that variable*; this rule can be used as a guideline when developing applications. This is what have been done with the *MM* application, which shows both an energy and a performance improvements when shared data are cached. This application, indeed, uses many shared variables as read-only objects (the source operands). Moreover, the variables accessed for writing (the result matrix slices), even if shared in principle, are actually used by just one processor. This is because each processor is in charge of computing a mutually exclusive portion of the result.

In Figure 13 are reported the performance, power, energy, and EDP, for the MM application varying the matrix size. Results support the considerations used in writing the code; this application benefits the advantages provided by the cache-coherence protocols. In fact, the slices of the resulting matrix, even if shared in principle, are actually used by just one processor. Each core computes a mutually exclusive portion of the result.

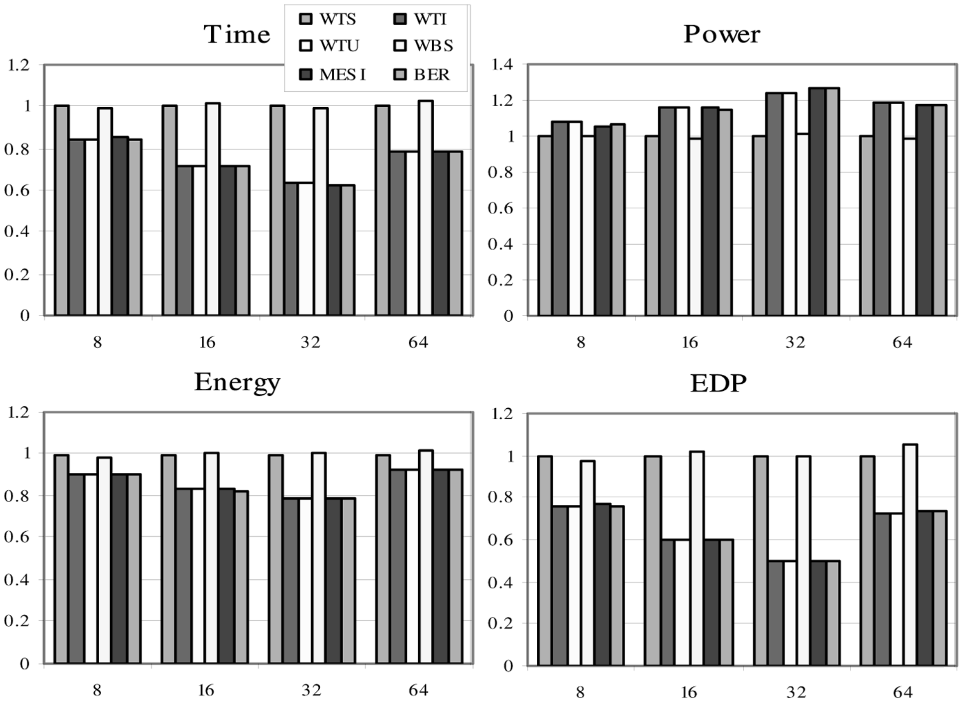


Fig. 13. Relative performance, power, energy, and EDP for the MM application for varying matrix size.

The resulting access pattern on the actual shared data is composed mainly of reads and, in this situation, the cache-coherent platforms obtain the best results. Given this access pattern, the difference between WT and WB policies is immaterial; since most of the execution time is spent on accessing shared variables, it is the cache coherency by itself which makes the difference, regardless of cache policy. Performance improvements and energy savings are around 30% (EDP reduction of 50%).

One last observation is that the advantages of the hardware cache-coherent solutions increase when larger matrices are used, because the cache is used more intensively. However, these improvements occur until the matrix size remains under a given limit, because when the data are too large, the number of the needed cache refills increase, hence, degrading the efficiency of the cache-coherence techniques.

Unfortunately, the nice sensitivity of MM to cache-coherence does not hold for generic parallel applications, that do not follow the above-mentioned guideline, such as the FFT and LU kernels, for which the impact of the cache coherence are less effective. Figures 14 and 15 show the results for the FFT application varying the vector size and for the LU application varying the matrix size.

Both these applications perform their computation using shared data, but shared variables are accessed with several different patterns and it becomes unlikely that many “good” sequences do exist. We notice the same behavior observed for synthetic benchmarks, namely, the dominance of the cache policy

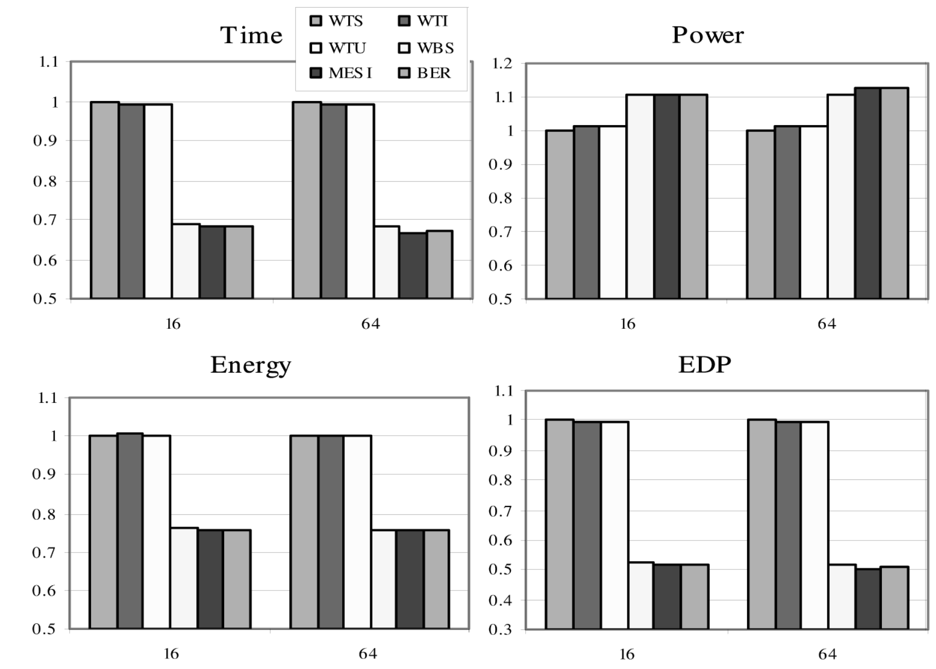


Fig. 14. Relative performance, power, energy, and EDP for the FFT application for varying vector size.

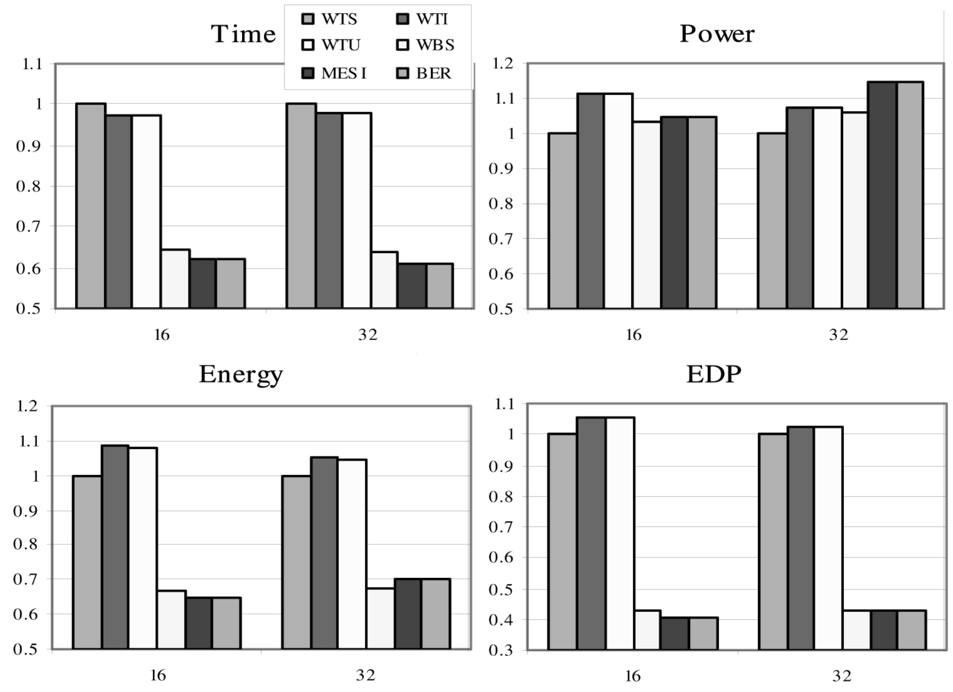


Fig. 15. Relative performance, power, energy, and EDP for the LU application for varying matrix size.

with respect to the CC protocol; the advantage of avoiding memory writes as for private as well for shared data is clear from the plot for both the applications. This advantage well amortize the overhead of larger tags and of the increased complexity of the processor. Notice that energy, performance, and EDP improvement are significant (over 50% in some cases).

Power figures deserve a specific comment. We notice that in many cases power for WB schemes increases; this is because of the fact that a significant contribution to energy reduction is because of an even larger improvement in performance. Since power is *average power*, and thus the ratio of energy and performance, power increase. In practice, the whole system is working with a higher efficiency (i.e., with a higher average cost for the individual operations), removing CPU stall cycles, which unfavorably consume energy without doing any useful work.

6. SUMMARY AND CONCLUSIONS

This paper compares three different approaches for guaranteeing cache-coherence for MPSoC architectures, namely, snoop-based coherence, a software-based approach which prevents caching of shared data, and an OS-based approach. All three solutions are, in principle, viable for tightly constrained architectures, because they have limited overhead. Our findings can be summarized as follows:

- The OS-based approach has excessively high cost. This underlines the critical need for software libraries carefully optimized for a target hardware platform. General-purpose libraries, such as those provided in the RTEMS system are not a practical alternative in a highly performance- and power-constrained context.
- Comparison between different snoop-based cache coherency schemes reveals a strong sensitivity to the cache write policy more than the specific coherency protocol. Write-back schemes appear to be more efficient, despite increased hardware complexity of cache-coherency support.
- Hardware-based cache-coherence appears to be competitive in terms of performance with respect to basic architectures with no hardware support, but it has significant power cost when coherence traffic grows, thereby casting some doubts on its viability when power constraints become tighter and the degree of multiprocessing increases. This is because of the fact that the relative power and energy costs for an MPSoC are quite different from classical multiprocessors where the bus and shared memories are off-chip. More specifically, the speed ratio between cache hit and shared memory access is less than an order of magnitude and the consumption for accessing fast and power-hungry cache memories is larger than that for for on-chip bus and slower shared memories. Hence, it is harder to amortize power the cost of caches when many redundant accesses are performed by the snoop devices.
- Light-weight schemes, which avoid caching of nonshared data, are generally energy efficient. Moreover, they enjoy the advantage of scalability. Snoop-based cache coherency is viable only on a bus-based system and extending

hardware cache coherency to more complex MPSoC architectures with multihop, scalable high-bandwidth interconnection networks entails very significant challenges. On the contrary, careful allocation of shared variable in distributed shared memories is feasible even when the architectures become highly distributed. Clearly, programming support for memory mapping and allocation is a requirement in this case.

- For a specific access pattern, i.e., read-dominated access to shared data, hardware cache coherency has very significant advantages with respect to the noncacheable shared data solution. In this case, caching enables large speed-ups (resulting in a better energy efficiency) and cache-coherency guarantees that program correctness is preserved in case of rare, or unpredictable, writes. This read-dominated behavior is definitely possible in real applications (think, for example, to read accesses to a slowly varying coefficient array, or to compression table in a filter or compressor, which operates on many parallel frames).

Looking forward, our analysis indicates that a configurable, hybrid solution may be the optimal choice for shared-memory MPSoCs. We envision an architecture where memory regions can be declared as not cacheable, while snoop-based cache coherency is provided for cacheable regions. This approach may open the way for scalable large-scale MPSoC architectures, where noncacheable memories are used for communication between cache-coherent clusters. Defining a programming model and developing efficient communication APIs for these single-chip NUMA machines is an interesting and open research area.

REFERENCES

Ackland, B. et al. 2000. A single chip, 1.6 billion, 16-b mac/s multiprocessor dsp. *IEEE Journal of Solid State Circuits 35*, 3 (Mar.).

AMBA Home Page. http://www.arm.com/products/solutions/AMBAHomePage.html.

ARM11 MPCore. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

Benini, L. et al. 2001. A power modeling and estimation framework for vliw-based embedded systems. In *Proceedings of PATMOS'01*. 26–28.

Carter, J. et al. 1999. Impulse: building a smarter memory controller. In *Proceedings of HPCA'99*. 70–79.

Chinosi, M., Zafalon, R., and Guardiani, C. 1998. Automatic characterization and modeling of power consumption in static rams. In *Proceedings of ISLPED'98*. 112–114.

Culler, D., Gupta, A., and Singh, J. 1997. *Parallel Computer Architecture: A Hardware / Software Approach*. Morgan Kaufmann Publishers. San Francisco, CA.

Cumming, P. 2003. *The TI OMAP Platform Approach to SoC*. Kluwer Academic Publ, Boston, MA.

Ekman, M., Dahlgren, F., and Stenström, P. 2002a. Evaluation of snoop-energy reduction techniques for chip-multiprocessors. In *Proceedings of WDDD-1*.

Ekman, M., Dahlgren, F., and Stenström, P. 2002b. Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of ISLPED'02*. 243–246.

Geppert, L. 2005. Sun's big splash [niagara microprocessor chip]. *IEEE Spectrum 42*, 1 (Jan.), 56–60.

Grammatikakis, M., Coppola, M., and Sensini, F. 2003. Software for multiprocessor networks-on-chip. In *Networks on Chip*. Kluwer Academic Publishers, Boston, MA. 281–303.

Gries, M. 2000. The impact of recent dram architectures on embedded systems performance. In *Proceedings of the 26th Euromicro Conference*. 282–289.

Hill, M. 1998. Multiprocessors should support simple memory consistency models. *IEEE Computer 31*, 8 (Aug.), 28–34.

HONG, S. ET AL. 1999. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of HPCA'99*. 80–89.

KALLA, R., BALARAM, S., AND TENDLER, J. 2004. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro 24*, 2 (Mar.–Apr.), 40–47.

KELTCHER, C., MCGRATH, K., AHMED, A., AND CONWAY, P. 2003. The amd opteron processor for multiprocessor servers. *IEEE Micro 23*, 2 (Mar.–Apr.), 66–76.

LOGHI, M., ANGIOLINI, F., BERTOZZI, D., BENINI, L., AND ZAFALON, R. 2004. Analyzing on-chip communication in a mpsoc environment. In *Proceedings of DATE'04*. 752–757.

MARCHAL, P., BRUNI, D., GOMEZ, J., BENINI, L., PINUEL, L., CATTHOOR, F., AND CORPORAAL, H. 2003. Sdram-energy-aware memory allocation for dynamic multi-media applications on multiprocessor platforms. In *Proceedings of DATE'03*. 516–521.

MCNAIRY, C. AND BHATIA, R. 2005. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro 25*, 2 (Mar.–Apr.), 10–20.

MOSHOVOS, A., FALSAFI, B., AND CHOUDHARY, A. 2001. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proceedings of HPCA'01*. 85–97.

PHILIPS SEMICONDUCTOR. Philips nexperia platform.
http://www.semiconductors.philips.com/products/nexperia/home.

RICHARDSON, S. 2002. Mpoc: A chip multiprocessor for embedded systems. *HP Technical Report, HPL-2002-186*.

RTEMS HOME PAGE. http://www.rtems.com.

SALDANHA, C. AND LIPASTI, M. 2003. Power efficient cache-coherence. *High performance memory systems*. Springer-Verlag, New York. 63–78.

SILICONHIVE. http://www.silicon-hive.com.

SINGH, J., WEBER, W., AND GUPTA, A. 1992. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News 20*, 1 (Mar.), 5–44.

SOFTWARE ARM. http://www.g141.com/projects/swarm.

STENSTRÖM, P. 1990. A survey of cache-coherence schemes for multiprocessors. *IEEE Computer 23*, 6 (June), 12–24.

TARTALJA, I. AND MILUTINOVIC, V. 1997. Classifying software-based cache-coherence solutions. *IEEE Software 14*, 3 (Mar.), 90–101.

TOMASEVIC, M. AND MILUTINOVIC, V. 1994. Hardware approaches to cache coherence in shared-memory multiprocessors. *IEEE Micro 14*, 5–6 (Oct.–Dec.), 52–59.

Received February 2005; accepted June 2005