Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs: Snoop-Based Cache Coherence vs. Software Solutions

Mirko Loghi, Massimo Poncino

Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs: Snoop-Based Cache Coherence vs. Software Solutions

Mirko Loghi
Dipartimento di Informatica
Università di Verona
37134 Verona, Italy
loghi@sci.univr.it

Massimo Poncino
Dipartimento di Automatica e Informatica
Politecnico di Torino
10129 Torino, Italy
massimo.poncino@polito.it

Abstract

Shared memory is a common interprocessor communication paradigm for single-chip multi-processor platforms. Snoop-based cache coherence is a very successful technique that provides a clean shared-memory programming abstraction in general-purpose chip multi-processors, 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.

1. Introduction

The rapid advances of silicon technology have made it possible to build small- to medium-scale single-chip multiprocessors. These devices can be coarsely distinguished in two classes: *general purpose chip multiprocessors* [1, 2, 3] (GPCMs) which integrate a small number of advanced processor cores (e.g. Itanium 2, UltraSPARC) and large caches in a tightly connected cluster, and *Multi-Processor Systems-on-Chip* (MPSoC) [4, 5, 6, 7, 8], which contain simpler cores and many application-specific heterogeneous coprocessors, embedded memories and peripherals. These two classes have completely different application targets and design constraints. GPCMs target the high-end server market and are also often used as building blocks for large scale supercomputers. In contrast, MPSoC are targeted for embedded applications (multimedia, video, graphics) in tightly cost- and power-constrained markets (e.g., smart phones, home entertainment centers, etc.).

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. [9, 10] 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., [6]), other devices on the market rely on non-cache-coherent architectures (e.g., [4]). 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 which are technology-homogeneous (i.e., all obtained from characterization in the same $.13\mu m$ technology). We compared three alternative approaches to cache-coherence: in the first one (*hardware-based*), cache coherence is imposed by a specific device implementing a snoopy protocol; in the second one (*software-based*), 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 (based on message passing).

Our analysis allows us to derive some interesting and non-trivial 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.

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. Widely speaking, approaches for solving the cache coherence problem in multiprocessor systems fall

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

into two major classes: hardware-based approaches, and software-based ones. 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 [11, 13, 12]; software-based cache coherence solutions are reviewed in [14].

Hardware-based cache coherence solutions (called cache coherence *protocols* hereafter) can be further classified according to two orthogonal dimensions [11]:

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, due to 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 requires 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 [15], 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 do exist, a cache access is avoided, thus achieving significant energy savings. Another approach is *serial snooping* [16], 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 multi-chip SMPs; they have been evaluated on GPCMs in [9], 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 [10].

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 schemes require quite high computational and/or hardware overhead, and assume non-realistic software architectures. For example, in [9, 10] 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 1 shows the architectural template of the multiprocessor simulation platform used in this work, called MPARM [17]. 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.

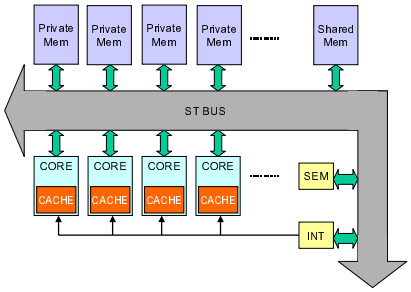


Figure 1. Hardware Architecture.

The processor cores are modeled by means of an adapted version of a GPL-licensed Instruction-Set-simulator (ISS) called SWARM [18], written in C++ and embedded into a SystemC wrapper. Each ISS contains his own 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. The platform is entirely described in SystemC at the signal level, except the ISS which performs cycle-accurate simulations of the cores.

The platform is configurable, and it allows to specify several parameters, such as the type of interconnect (AMBA vs. ST-Bus), the number of processing elements, and the cache and memory parameters (cache size, line size, cache type, memory size and latency, etc.)

A port of a real-time operative system called RTEMS [20] is available for the platform. RTEMS offers support for multiprocessing, and native calls for process communication and synchronization. Inter-process and inter-thread communication in RTEMS rely on *message queues*; threads communicate using the a `send` primitive to put data into the queue, and a `receive` primitive to retrieve data, both provided by RTEMS as system calls.

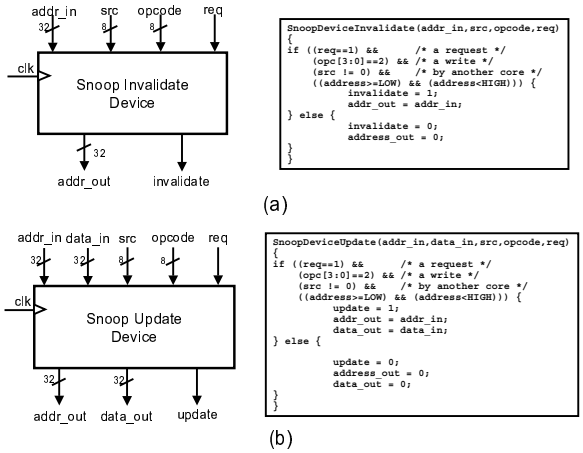
Concerning power analysis, the platform provides accurate power models associated to each component. All models are cycle accurate, and have been characterized on a 0.13 μm technology by STMicroelectronics and validated on silicon implementations of the various components. For the processor cores we have used an instruction-based power model (see [21] for details). For the memories (both caches and private memories) we have used an analytical model derived from the work presented in [22], whereas the power model for the STbus interconnect has been taken from [23].

3.1. 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 a hardware coherence support based on a write-through policy, which comes into two variants: one based on an invalidate policy (*Write-Through Invalidate, WTI*), the other based on an update policy (*Write-Through Update, WTU*).

The devices are quite similar: 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 taken (invalidate or update). Notice that the update device is slightly more complex. It features extra I/O interface for the data, since in this case we are writing a value to the updated locations.




```
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;
}
}
```

```
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;
}
}
```

Figure 2. Operations of the Snoop Device for the Invalidate (a) and Update (b) Policies.

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. When a write operation is flagged, the corresponding action is performed, i.e., invalidation for the WTI policy, rewriting of the data 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 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. In our simulation model, synchronization between the core and the snoop device is handled using shared variables. During a clock cycle only one of them can access the cache memory; therefore the other one must be locked, waiting for the cache accessibility.

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* [12]. The following sub-sections describe these three schemes. For each scheme, its corresponding programming model will be illustrated through a simple example, namely, a 1-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 3 shows a typical textbook pseudocode for our working example. The use of the `shared` variable `value` is guaranteed to be coherent. However, the programmer must use a synchronization variable (a semaphore, in the example) to regulate access to the data.

```
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);
    }
}
```

Figure 3. Producer-Consumer Pseudo-Code with 1-item buffer.

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 [14]. 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.

In our example, the code differs only marginally from the one in Figure 3, namely, the `shared` keyword used on a variable automatically implies the fact that it cannot be cached.

4.3. OS-based Coherence

The third scheme consists of leaving to the OS do the task of guaranteeing coherence, through its IPC abstraction offered by its API. Specifically, RTEMS offers a communication infrastructure based on *message queues* [20], 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 4, 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.

```
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);
    }
}
```

Figure 4. Producer-Consumer Pseudo-Code with 1-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 which 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 to have exclusive access to the shared data. Unlike the previous case, multiple writes before one read and multiple reads before one write are possible Also in this case, each process (reader or writer) 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.

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 [24].

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.
As a first experiment, we compared the three coherence schemes on the producer-consumer application (`PC2,2,16`). Figure 5 shows performance, energy and power results of the various coherence schemes. *WTI* (*WTU*) denotes hardware-based coherence using invalidate (update) protocol, *SW* denotes software-based coherence and *OS* denotes coherence imposed by OS communication primitives.

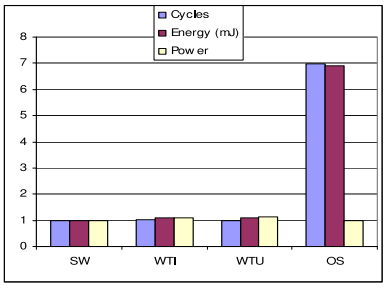


Figure 5. Relative Energy and Performance for the Producer-Consumer Application.

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 due to the OS is generally to be expected, but in this case it appears to be quite significant. Since the penalties due to the OS are very similar for all the applications used and for all the platform configurations, results relative to OS-based coherence will not be considered further.

5.2.1. Synthetic Benchmarks. For the producer-consumer benchmark, 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, due to the non-cacheability of its address space, this variations have the same impact on system performance and power consumption for the cache coherent solutions as well as for the non-coherent one. The WTU solution results slightly more efficient than the SW one in terms of execution time; however, it also has larger energy consumption. The WTI scheme has a worse behavior for the energy consumption and has an execution time which can be worst or slightly better than the SW one, but always worse than the WTU one.
The second set of experiments refers to the `RW` application, relative to the case of 3 readers and 1 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), 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 which benefit from this fact, but also sequences which 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 an 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 may be wasted.

Figure 6 shows performance, power, energy, and EDP, where the relative speed of readers and writers has been set to 1.

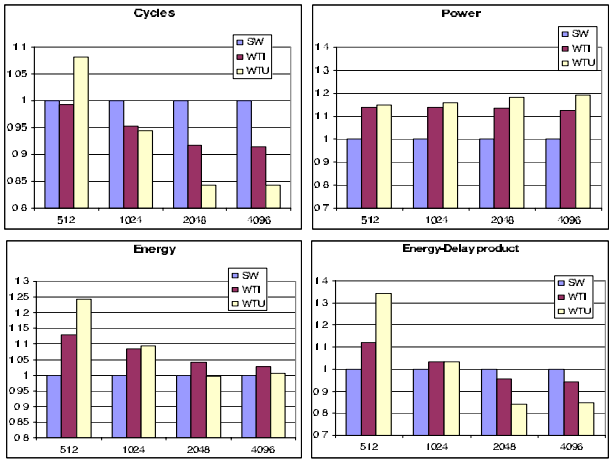


Figure 6. Relative Performance, Power, Energy and EDP for the RW Application for Varying Cache Sizes.

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 to cache shared data, as the experiments show. By increasing the cache size, we can observe an improvement of performance for both the HW solutions, until the gain saturates. The saturation is due to the point after which the cache is bigger than the whole 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 the synchronization, and the usefulness of the cache coherent scheme tends to decrease. This is shown in Figure 7, 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 1KB bytes case too (the object is too large for the cache used).

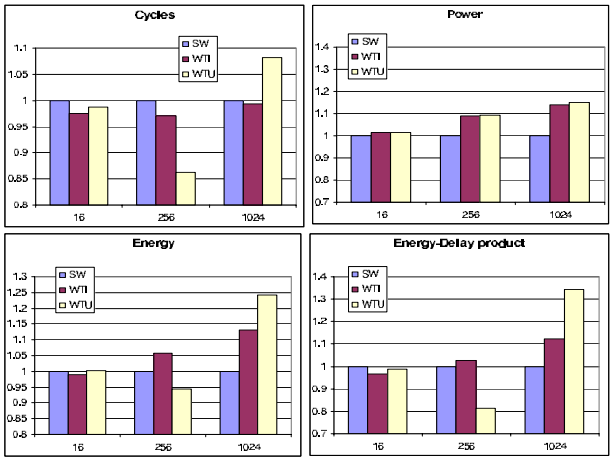


Figure 7. Relative Energy, Performance, Power and EDP for the RW Application for Varying Buffer Sizes.

The impact of the speed ratio between readers and the writers on the various metrics is also interesting. In fact, it directly affects the access patterns on the shared memory and, as a consequence, on data caches. If the writer is faster than the readers, there will be many consecutive writes on the shared object before a read happens. As mentioned above, this is the most negative condition for a cache coherent architecture.

This explain why the WTU solution is ineffective when the speed ratio equal to 0.1. The application with speed ratio equal to 10 is also penalized. Again, this occurs because of the unfavorable access pattern. The quicker reader finishes its work when the writer is still running. Thus, the final section of the access sequence is again composed of writes only.

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.

For applications that are not written with the optimization of the access patterns to shared variables in mind, the results are less effective. Both the *FFT* and the *LU* applications use shared data to perform their computation. Such data are accessed with several different patterns and it is unlikely that a

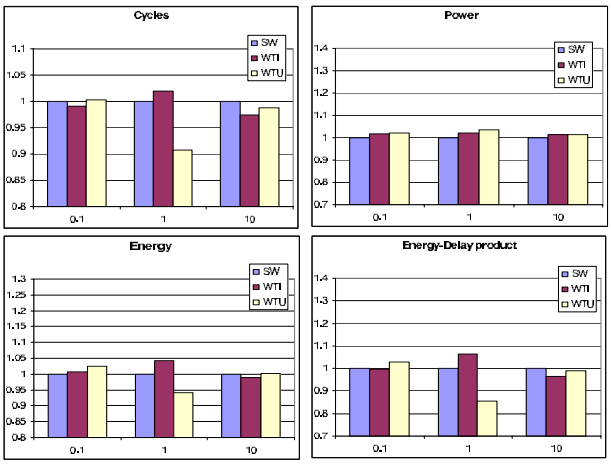


Figure 8. Relative Energy, Performance, Power and EDP for the RW Application for Varying Execution Speed Ratios.

"good" pattern does exist. Therefore both applications show slightly better performance, but, at the same time, a higher energy consumption, due to the energy wasted to keep the data synchronized and consistent into different caches.
Concerning power, we notice that power increases even when energy is decreased when cache coherence is involved. This is because when the energy decreases the performance increase even more, and their ratio increases. In practice, the whole system is working with a higher efficiency, removing CPU stall cycles which unfavorably consume energy without doing any useful work. Conversely, when enforcing coherence wastes energy, performance are impacted less significantly, because the high speed of the cache memories allows to recover many of the cycles wasted to keep data consistence.

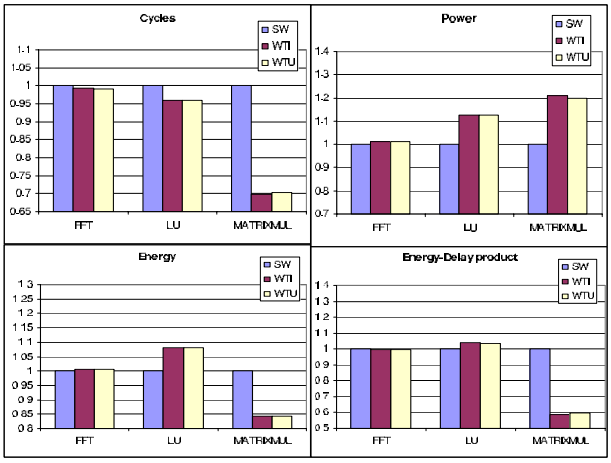


Figure 9. Relative Energy, Performance, Power and EDP for the Application Kernels.

6. 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 analysis reveals that the OS-based approach has excessively high cost. Hardware-based cache coherence appears to be competitive in terms of performance, 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 grows. Light-weight schemes which avoid caching of non-shared data are scalable and energy efficient, but they imply maximum programming effort.

Acknowledgements

We wish to thank Prof. Luca Benini from Università di Bologna for the helpful suggestions on the experimental analysis.

References

[1] "Broadening the Reach of the Intel Itanium 2 Processor Family," Technical White Paper, www.intel.com/ebusiness/pdf/prod/itanium/wp_reach.pdf
[2] M. Tremblay, J. Chen, S. Chaudry, A. Conigliaro, S.-S. Tse. "The MAJC Architecture: A Synthesis of Parallelism and Scalability,", *IEEE Micro*, Vol. 20, No. 6, Nov.-Dec. 2000, pp 12-25.
[3] J.M. Tendler, J.S. Dodson, J.S. Fields Jr., H. Le, B. Sin-Haroy. "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, Vol. 46, No. 1, January 2002.
[4] P. Cumming "The TI OMAP Platform Approach to SoC," in *Winning the SOC Revolution*, Kluwer Academic Publishers, 2003.
[5] S. Richardson, "MPOC: A Chip Multiprocessor for Embedded Systems,", HP Technical Report, HPL-2002-186, July 2002.
[6] B. Ackland et al., "A Single Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP," *IEEE Journal of Solid State Circuits*, Vol. 35, No. 3, March 2000.
[7] Philips Semiconductor, "Philips Nexperia Platform", www.semiconductors.philips.com/products/nexperia/home S. Dutta, R. Jensen, A. Rieckmann,
[8] M. Grammatikakis, M. Coppola, F. Sensini, "Software for Multiprocessor Networks-on-Chip," *Networks on Chip*, Kluwer Academic Publishers, pp. 281–303, 2003.
[9] M. Ekman, F. Dahlgren, P. Stenström "Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors," *ISCA'02*, May 2002.
[10] M. Ekman, F. Dahgren, P. Stenström, "TLB and Snoop Energy-Reduction Using Virtual Caches in Low-Power Chip-Multiprocessors," *ISLPED'02*, August 2002, pp. 243–246.
[11] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, June 1990, pp. 12–24.
[12] D.E. Culler, A. Gupta. J.P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach* Morgan Kaufmann Publishers, 1997.
[13] M. Tomasevic, V. M. Milutinovic, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors," *IEEE Micro*, Vol. 14, No. 5–6, pp. 52–59, October/December 1994.
[14] I. Tartalja, V. M. Milutinovic, "Classifying Software-Based Cache Coherence Solutions," *IEEE Software*, Vol. 14, No. 3, pp. 90–101, March 1997.
[15] A. Moshovos, B. Falsafi, A. Choudhary, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers", *HPCA'01* January 2001, pp. 85-97.
[16] C. Saldanha and M. Lipasti, "Power Efficient Cache Coherence", *High Performance Memory Systems*, Springer-Verlag, 2003, pp. 63–78.
[17] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, "Analyzing On-Chip Communication in a MPSoC Environment", *DATE'04*,February 2004, pp. 752–757.
[18] Software ARM, www.g141.com/projects/swarm.
[19] ARM Ltd., www.arm.com/products/solutions/AMBAHomePage.html
[20] RTEMS home page, www.rtems.com.
[21] L. Benini et al. "A power modeling and estimation framework for VLIW-based embedded systems," *PATMOS'01*, October 2001, pp. 26–28.
[22] M. Chinosi, R. Zafalon, C. Guardiani, "Automatic Characterization and Modeling of Power Consumption in Static RAMs," *ISLPED'98*, Aug. 1998, pp. 112–114.
[23] A. Bona, V. Zaccaria, R. Zafalon, "System-Level Power Modeling and Simulation of High-End Industrial Network-on-chip", *DATE'04*,pp. 318–323.
[24] J. P. Singh, W.-D. Weber, A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", *Computer Architecture News*, Vol. 20, No. 1, pages 5-44, March 1992.