Tuning Data Replication for Improving Behavior of MPSoC Applications

O. Ozturk, M. Kandemir, M. J. Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16802, USA
{ozturk, kandemir, mji}@cse.psu.edu

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK
ikolcu@umist.ac.uk

ABSTRACT

Maintaining cache coherence can be very costly for on-chip multiprocessors from an energy perspective. Observing this, we propose a compiler-directed strategy that replicates array data in cache memories of its potential consumer processors at the time the data is brought from off-chip memory. The goal is to eliminate the energy costs associated with bus snooping without negatively impacting overall performance. Our strategy can perform a much better job as compared to static replication strategies, where each array element is replicated based on the same fixed policy.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *compilers, optimization.*

General Terms

Languages, Performance.

Keywords

MPSoC, Power Consumption, Data Replication, Cache Coherence, Optimizing Compiler.

1. INTRODUCTION

As the applications ported into system-on-a-chip (SoC) architectures become more and more complex, it is becoming extremely important to have sufficient compute power on the chip. One way of achieving this is to put multiple processor cores in a single chip. This *multiprocessor-system-on-a-chip* (MPSoC) architecture has several advantages over an alternate strategy that puts a more powerful and complex processor in the chip. First, this architecture is very suitable for array-intensive applications from image/video processing domain. Second, since the processors in an MPSoC are in general simple (i.e., with no speculation/predication logic for example), this architecture is also energy efficient. Third, the design of an on-chip multiprocessor composed of multiple simple processor cores is simpler than that of a complex single processor system [5,8]. This simplicity also helps reduce the time spent in verification and validation [8]. Fourth, MPSoCs have lower link lengths and delays.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'04, April 26-28, 2004, Boston, MA, USA.
Copyright 2004 ACM 1-58113-853-9/04/0004…$5.00.

Current trends and future projections in industry also signal a big shift towards MPSoC architectures. For example, Sun's new H-series line of microprocessors will debut in 2005, with one variant expected to offer the equivalent of a 32-way symmetric multiprocessor system on a chip. Interestingly, the formal announcement for this on-chip multiprocessor came just one day after IBM announced it would ship in 2004 its Power5, a dual-core 64-bit server processor running two threads on each die. Intel Corporation has also said it will roll out a dual core version of its 64-bit Itanium processor in 2005. Note that Intel already ships dual-threaded 32-bit Pentium4 and Xeon processors that use a single core [10]. These trends clearly indicate that future systems will invest more and more on MPSoC type of architectures.

One of the most critical issues in an MPSoC environment is to optimize its memory system behavior. This is because making frequent off-chip data accesses can both degrade overall performance and result in significant power consumption. To avoid these, locality of data accesses should be optimized as much as possible; that is, most of the time, when a processor requests a data item, it should be able to find it in its local cache – this helps maximize performance and minimize power consumption.

In many array-intensive MPSoC applications, on-chip processors share some data. For a shared data item (e.g., an array element), as far as its location at the time of access is concerned, there are three possible scenarios – (1) the item is in the off-chip memory. This is the worst case as indicated earlier. (2) the item is in the local cache of some other processors (i.e., not the requester). While this is better than the first case, accessing a non-local cache can consume both execution cycles and power. (3) the item is in the local cache of the requester. This is the best possible case. Consequently, one of the objectives of an optimizing compiler targeting array-intensive MPSoC applications should be increasing the local cache hits for the shared data.

A naïve way of implementing this idea would be replicating each data item in each on-chip cache (when the data item is brought from off-chip memory). The obvious problem with this approach is that not all data items are shared. Moreover, even the ones that are shared are not shared by all on-chip processors. Replicating such unshared or narrowly shared data can reduce effective cache capacity available and hurt overall performance. Therefore, for each application, there should be some optimal amount of replication that gives the best performance and energy behavior.

One might argue that the right amount of sharing would be automatically achieved by a cache coherence protocol that could be used for MPSoC (e.g., a MESI-like protocol). However, since such a conventional coherence protocol may be very costly to maintain (from a power consumption perspective) for an MPSoC based architecture, our objective in this study is to eliminate it if it is possible to do so. We attempt to achieve this by replicating data across the processors (that are likely to share it) when it is brought from off-chip memory to the MPSoC. We also attach a tag to the cache line indicating in which caches it is replicated. At runtime, the processors do not snoop bus activity (this can save significant energy). Instead, whenever a write to a shared data occurs, we invalidate the copies in the other processors (that share the data in question) directly. In this approach, the most important problem is to decide the right amount of replication, i.e., when the data is brought from off-chip memory to on-chip, we need to decide where (in which caches) to replicate it so that when some other processor requires the same data, it does not need to go to the off-chip memory (note that this saves execution cycles as well as energy).

In this paper, we make two main contributions. First, we discuss and evaluate several *static strategies* that implement different data replication algorithms and compare them to a classical coherence-based mechanism that uses MESI. Our proposed strategies outperform the classical coherence-based strategy from an energy consumption viewpoint. Note that maintaining coherence is very easy in our case since we know exactly which caches share the data (this is exactly the set of caches over which the data is replicated). However, the proposed strategies are far from being optimal, and do not perform well from a performance viewpoint. The second contribution of this paper is a *compiler-directed adaptive data replication algorithm*, whereby the set of caches to replicate the data over is decided in an adaptive manner, considering the (compiler-analyzable) access pattern of the data in question.

Adve et al [1] use an analytical model to compare the performance of compiler-directed and directory-based schemes. Lim and Yew [6] propose a strategy that enforces cache coherence by prefetching the up-to-date data corresponding potentially stale references from the main memory. Choi and Yew [3] study compiler support for cache coherence in large-scale multiprocessor machines. In contrast to these studies, our target architecture is a bus-based MPSoC, and our main goal is optimizing energy consumption. Note that due to their high implementation costs from the energy perspective, these techniques and their variants may not be very suitable to be employed in an embedded MPSoC based environment.

We organize this paper as follows. In Section 2, we discuss the target architecture we consider in this work. In Section 3, we summarize important characteristics of the MESI coherence protocol. This is because we compare our techniques against a MESI-based architecture. In Section 4, we introduce our static strategies. In Section 5, we discuss our adaptive replication strategy. Finally, in Section 6, we conclude the paper with a summary of our major contributions and a discussion of planned future work on this topic.

2. MPSoC ARCHITECTURE

In this paper, we focus on the MPSoC architecture shown in Figure 1. This architecture contains multiple processors (each with its own instruction and data caches), and an inter-processor synchronization and clock logic. This is a shared memory architecture; that is, all inter-processor communication occurs through reading from and writing into the off-chip memory (also shown in the figure). A bus-based on-chip interconnect is used to perform inter-processor synchronization. Such synchronization is necessary for the processors to get synchronized at the beginning and end of each loop nest they execute. The processors we assume in this study are single-issue, five-stage pipelined architectures without any complex branch prediction or data speculation/prediction logic. This brings an important side-advantage in terms of execution time predictability since it is easier to predict execution time with simple processors without sophisticated prediction/speculation logic (a big plus in real-time embedded environments). Also, each processor can operate independently from each other, and processors engage in synchronization/communication only to maintain data integrity during parallel execution.

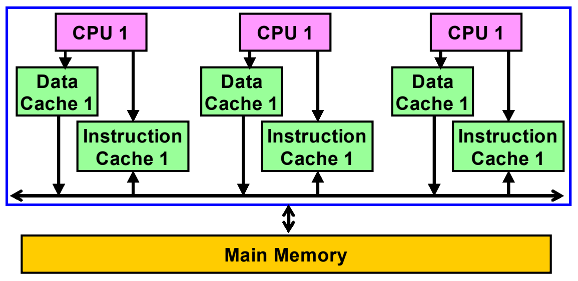

Figure 1. The MPSoC architecture considered in this study.

An array-based application is executed on our on-chip multiprocessor architecture by parallelizing its loop nests. Specifically, each loop is parallelized such that its iterations are distributed across available processors. An effective parallelization strategy should minimize the inter-processor data communication and synchronization (i.e., coherence activity should be minimized). In other words, ideally, each processor should be able to execute independently without synchronization or communication. However, in many cases, data dependences that occur across loop iterations prevent coherence activity-free execution. While in theory we can accommodate any loop-level parallelization strategy, in this study we use a data locality-oriented approach that operates as follows. Given a program, our approach handles each loop nest one-by-one. In optimizing a loop nest, it first uses loop transformations such as loop permutation, iteration space tiling, and scalar replacement, and places the loops with high data reuse into innermost position in the nest to the extent possible. In doing so, it creates outer loops that are dependence-free. After that, the outermost loop that does not carry any data dependence is parallelized by distributing its iterations across the processors. In distributing loop iterations, each processor is assigned a set of successive iterations. It should also be observed that such a parallelization strategy is very suitable from a cache locality perspective as well. The details of our parallelization strategy are beyond the scope of this paper.

3. MESI PROTOCOL

MESI is an invalidation-based protocol for write-back caches used in high-end multiprocessor machines. In this protocol, a cache line can be in one of four states: modified (M) or dirty, exclusive-clean (E), shared (S), and invalid (I); and each cache maintains the state information for all the lines it currently has. The state I means that the line is invalid. M indicates that only the cache under consideration has a valid copy of the line, and the copy in the main memory is stale. E means that only the cache under consideration has a copy of the line, which is the same as the corresponding copy in the main memory. Finally, S means that potentially two or more processors have this line in their caches in an unmodified form. Note that state E helps reduce bus traffic for sequential portions of the code where data is not shared. More detailed information on MESI protocol can be found elsewhere [4].

It should be noted that MESI is a snoop-based protocol, which means that processors continuously observe the traffic on the bus to take appropriate coherence actions. Therefore, as far as energy consumption is concerned, there are two major cost components: 1- snooping the bus for every action consumes energy and 2- executing the protocol itself consumes energy. Moshovos et al [7] propose a strategy that reduces the energy cost of tag checking in a symmetric multiprocessor (SMP) environment. In the rest of this paper, we make a case for a compiler-initiated scheme for array-based applications for which the compiler can statically derive accurate data access pattern information for each processor, taking into account the parallelization information available for each loop nest.

4. STATIC REPLICATION STRATEGIES

In this section, we present our static strategies for deciding how to replicate data read from off-chip memory. These strategies are called *static* because the replication strategy is fixed throughout the execution, and is applied to all data items (array elements) uniformly.

In *no-replication scheme* (NR), the data is placed only in the cache of the processor that requested it; that is, no replication is performed. Obviously, one can expect this strategy to be useful only when there is very little sharing. In *neighborhood replication strategy* (BR), the data is placed in the requester's cache as well as those of its left and right neighbors (if any). Finally, in *all replication scheme* (AR), the data brought from off-chip memory is replicated in all processors in the MPSoC.

While NR, BR, and AR provide slight energy improvements over the conventional strategy, they are not extremely successful in reducing energy consumption. In order to support this conclusion, we experiment with an optimal scheme, which, using an oracle, determines the best replication strategy each time a data is to be brought from off-chip memory to the MPSoC. This optimal strategy performs much better than the static strategies in terms of both energy saving and execution cycle improvement, which indicates one can potentially do much better than the static strategies. The next section discusses a compiler-based adaptive replication strategy along this direction. It should be noticed that the purpose of any adaptive strategy should be tuning the degree of replication according to the degree of sharing for as many array elements as possible.

5. COMPILER-BASED DATA REPLICATION

The static replication strategies do not perform very well, mainly because of two reasons. First, for the best energy consumption behavior, different data items (array elements) demand different replication strategies. This is to be expected since different data may exhibit very different sharing behavior. For example, while a data item might be highly shared (and could thus benefit from replication over all processors), for some other data item it might be sufficient to replicate it in only two neighboring processors. The second drawback of the static schemes is that the required replication behavior of the same data item may change in different phases of the execution of the same application. These two factors, combined, prevent the static schemes from achieving the best energy consumption behavior. These factors also indicate that an adaptive scheme that tunes the replication policy according to data access pattern (i.e., customizes replication for each data item) can potentially be very successful in practice.

The adaptive strategy proposed in this work is based on compiler analysis and takes advantage of locality of data references; that is, at a given time, most of data references accumulate in a specific memory region. In addition, it also exploits parallelization information that is available after the loop nests in the application are parallelized. For example, consider the following nested loop (written in a pseudo-language) that accesses three different arrays:

```
for i = 1..n
    for j = 1..m
        x[i][j] = y[j][i] + z[j]
```

Assuming that only the outer loop is parallelized in this nest, it is easy to see that none of the elements of arrays x and y is shared, whereas all the elements of array z are shared by all processors used to execute the nest. Therefore, in this loop nest, an array element has a degree of sharing of either 1 (for x and y) or 8 (for z). Consequently, ideally, we should replicate an array element either in all processors or in no processor at all. Now, let us consider the following loop that accesses a single array:

```
for i = 2..n-1
    x[i] = (x[i-1] + x[i+1])/2
```

Assuming again that this loop is parallelized over all the processors and each processor executes a number of consecutive (loop) iterations, one can see that some elements of the array x are shared by neighboring processors, whereas some other elements are not shared at all.

It is to be noted that an optimizing compiler can analyze a given array-based code and for each (static) array operation (read or write) decide whether the data accessed through it are shared or not. Moreover, it can also determine (estimate) the number of processors that will share it at runtime. In other words, the compiler is in a very good position to determine the degree of sharing for array-based data. Then, based on the degree of sharing, it can set the degree of replication accordingly. However, it should also be noticed that a static array operation accesses (in general) multiple array elements and there might be cases where these array elements exhibit different degrees of sharing. Therefore, an important question now is how frequently such cases occur (note that it does not occur in the above code examples). Fortunately, for most applications, such cases do not appear frequently. Specifically, the average value (cross all applications) of the percentage of the time a static array reference accesses data that exhibit different degrees of sharing is around 2.26%. Consequently, there is not much danger in assuming that each static array access exhibit uniform sharing pattern throughout the execution.

Our compiler-initiated adaptive data replication strategy works as follows. The compiler first analyzes the code to be optimized and, for each array reference, identifies the type of data reuse. Also, for each array in the code, the compiler determines the set of elements that will be accessed (at runtime) by each processor. In our compiler implementation of the adaptive strategy, we take into account parallelization and data reuse information. More specifically, assuming that Ψ_i and Ψ_j represent the set of iterations that will be executed by two processors i and j (this is derived from the parallelization information), these two processors share an array element ω if and only if there are at least two loop iteration points $\lambda_i \in \Psi_i$ and $\lambda_j \in \Psi_j$ such that both these iterations access ω. Therefore, the set of array elements (from array x) that are shared by i and j can be expressed as:

$$\Xi\,(i,j,x) = \{\omega \mid \exists\ \lambda_i \in \Psi_i \text{ and } \exists\ \lambda_j \in \Psi_j \text{ such that } Z_{x,i}(\lambda_i) = Z_{x,j}(\lambda_j)\},$$

where $Z_{x,i}(\lambda_i)$ and $Z_{x,j}(\lambda_j)$ give the array element accessed by iterations λ_i and λ_j, respectively, using an array reference. In our current implementation, we use the Omega Library [9] to enumerate/count the elements in this set, and we do this for every array in the code and every processor pairs i and j. The Omega Library is a set of C++ classes for manipulating integer tuple relations and sets. It is used in compiler research for dependence analysis, program transformations, generating code from transformations, and detecting redundant synchronization.

6. CONCLUSIONS AND FUTURE WORK

This paper presented several alternate strategies for managing data sharing in on-chip multiprocessors. Due to low power requirements, it may not be very efficient to employ a full MESI–like protocol in MPSoC-based environments. Our compiler-based adaptive strategy captures data sharing much better than the static strategies, which fix degree of replication for each array element at the same value. Our future work involves developing pure hardware-based adaptive replication strategies and comparing them with the compiler-based strategy. Work is also underway in modifying the MESI protocol itself to make it more energy efficient.

7. REFERENCES

[1] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. In Proc. 18th Annual International Symposium on Computer Architecture, pp. 298-308, May 1991.

[2] CACTI 3.0. http://research.compaq.com/wrl/ people/ jouppi/CACTI.html

[3] L. Choi and P. C. Yew. Compiler analysis for cache coherence: inter-procedural array data-flow analysis and its impact on cache performance. IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 9, Sept 2000, pp. 879-896.

[4] D. E. Culler and J. P. Singh. Parallel computer architecture: a hardware-software approach. Morgan Kaufmann, 1999.

[5] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multi-threading. IEEE Transactions on Computers, Special Issue on Multi-threaded Architecture, September 1999.

[6] H. B. Lim and P. C. Yew. Maintaining cache coherence through compiler-directed data prefetching. Journal of Parallel and Distributed Computing, Vol 53, No. 2, pp. 144-173, September 1998.

[7] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: snoop filtering for reduced energy consumption in SMP servers. In Proc. Symposium on High-Performance Computer Architecture, January 2001.

[8] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluating alternatives for a multiprocessor microprocessor. In Proc. the 23rd Intl. Symposium on Computer Architecture, pp. 66--77, Philadelphia, PA, 1996.

[9] W. Pugh. Counting Solutions to Presburger Formulas: How and Why. In Proc. the ACM SIGPLAN Conference on Programming Language Design, 1994.

[10] http://www.siliconstrategies.com/story/OEG20030225S0031