Exploiting On-Chip Data Transfers for Improving Performance of Chip-Scale Multiprocessors

G. Chen[1], M. Kandemir[1], I. Kolcu[2], and A. Choudhary[3]

[1] Pennsylvania State University, PA 16802, USA
[2] UMIST, Manchester M60 1QD, UK
[3] Northwestern University, Evanston, IL 60208, USA

Abstract. As compared to a complex single processor based system, on-chip multiprocessors are less complex, more power efficient, and easier to test and validate. In this work, we focus on an on-chip multiprocessor where each processor has a local memory (or cache). We demonstrate that, in such an architecture, allowing each processor to do off-chip memory requests on behalf of other processors can improve overall performance over a straightforward strategy, where each processor performs off-chip requests independently. Our experimental results obtained using six benchmark codes indicate large execution cycle savings over a wide range of architectural configurations.

1 Introduction

A natural memory architecture for an on-chip multiprocessor is the one in which each processor has a private (on-chip) memory space (in form of either a software-controlled local memory or a hardware-controlled cache). Consequently, assuming the existence of an off-chip (shared) memory, from a given processor's perspective, we have a two-level memory hierarchy: on-chip local memory and off-chip global-memory. One of the most critical issues in exploiting this two-level memory hierarchy is to reduce the number of off-chip accesses. Note that reducing off-chip memory accesses is beneficial not only from the performance perspective but also from the energy consumption viewpoint. This is because energy consumption is a function of the effective switching capacitance [2], and large off-chip memory structures have much larger capacitances as compared to relatively smaller on-chip memory structures.

In this paper, we focus on such a two-level memory architecture, and propose and evaluate a compiler-directed optimization strategy which reduces the frequency and volume of the off-chip memory traffic. Our approach targets array-intensive applications, and is based on the observation that, in an on-chip multiprocessor, inter-processor communication is cheaper (both performance and energy wise) than off-chip memory accesses. Consequently, we develop a strategy which reduces off-chip memory accesses at the expense of increased on-chip

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 271–278, 2003.

data traffic. The net result is significant savings in execution cycles. Our approach achieves its objective by not performing off-chip memory accesses based on data requirements of each individual processor but based on the layout of the data in the off-chip memory. In other words, the total data space (array elements) that will be required considering all processors is traversed (accessed) in a layout-efficient manner (instead of allowing each processor access off-chip memory according to its own needs). The consequence of this "collective" memory access strategy is that each processor gets some data which are, potentially, required by some other processor. In the next step, the processors engage in a many-to-many communication (i.e., they access each others local memories) and, after this on-chip communication step, each processor gets the data it originally wanted.

We implemented the necessary compilation techniques for this two-step optimization strategy using an experimental compiler, and collected results from this implementation. Experimental data obtained using six application codes clearly demonstrate the effectiveness of our strategy. In order to measure the robustness of our approach, we also conducted experiments with different architectural parameters (due to the space limitation, the results are omitted in this paper). Based on our experimental data, we believe that the proposed strategy is a much better alternative to a more straightforward strategy, where each processor performs independent off-chip memory accesses. Our experimental results indicate large execution cycle savings over a wide range of architectural configurations. The results also indicate that the proposed approach outperforms an alternate technique that employs classical locality-oriented loop optimizations.

2 Architecture Abstraction

In this paper, we focus on an architectural abstraction depicted in Figure 1. In this abstraction, the on-chip multiprocessor contains k CPU-local memory pairs. There is also an off-chip (shared) global memory. There are two buses: an on-chip bus that connects local memories to each other, and an off-chip bus that connects local memories to the off-chip global memory. The other components of the on-chip multiprocessor (e.g., the clock circuitry, ASICs, signal converters, etc.) are omitted as they are not the focus of this work. It should be mentioned that many architectural implementations described by prior research are similar to this abstraction.

In this architecture, a data item required by CPU_j ($1 \leq j \leq k$) can be in three different locations: local memory_j, local $\text{memory}_{j'}$ of $\text{CPU}_{j'}$ (where $1 \leq j' \leq k$ and $j' \neq j$), and the global memory. The performance of a given application executing on such an architecture is significantly influenced by the location of the data requested. Typically, from the viewpoint of a given processor, accessing its local memory is much faster than accessing the global memory (and, in fact, considering current trends, we can expect this gap to be widened in the future). As compared to accessing its own local memory, accessing the local memory of some other processor (called "remote local memory," or "remote memory" for

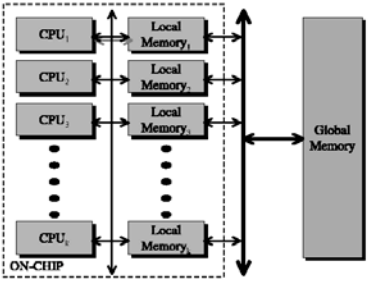

Fig. 1. On-chip multiprocessor abstraction.

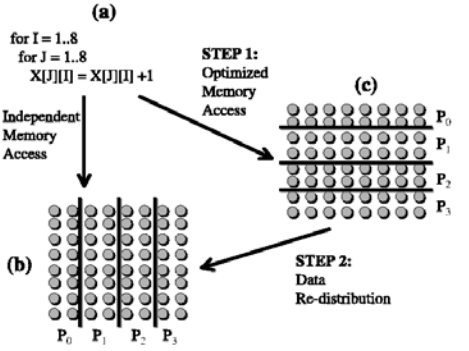

Fig. 2. Independent memory access vs. optimized memory access and data redistribution.

short) is also more costly; however, such remote memory accesses are still much less expensive in comparison to global memory accesses. Therefore, an important issue in compiling applications for running on this chip-scale multiprocessor environment is to ensure that most of data requests of individual processors are satisfied from their local memories or remote memories. The goal of this paper is to present and evaluate a compiler-directed optimization strategy along this direction. It should be emphasized that, in this paper, when we mention "on-chip communication," we mean remote memory access. Consequently, when we say "processors engage in all-to-all (or many-to-many) communication," we mean that they access each other's local memory. We also use the terms "off-chip memory" and "global memory" interchangeably.

3 Our Approach

Performance of an application executed on an on-chip multiprocessor is closely tied with the data access pattern. This is because data access pattern is the primary factor that determines how memory is accessed. Therefore, it is critically important that an application exhibits a good access pattern, in which data is accessed with temporal or spatial locality to the highest extent possible. Accesses with locality tend to frequently reuse the data in the same transfer unit (block), and this, in turn, reduces the number of off-chip memory accesses.

However, a straightforward coding of many array-intensive applications does not lead to good memory access patterns. As an example, consider the simple loop structure given in Figure 2(a). Assuming that we would like to parallelize the outer loop across four processors (P_0, P_1, P_2, and P_3) in our on-chip multiprocessor, Figure 2(b) shows the data access pattern exhibited when each processor performs off-chip accesses independently. It can be seen from this access pattern that, assuming a row-major array layout in memory, each processor touches only a small amount of sequential data. In other words, the spatial locality exhibited

by each processor is very low. To quantify this, let us conduct a simple calculation to see the memory access cost of this loop nest. Assuming that the granularity of the data transfers between the off-chip memory and on-chip memory is 4 array elements and that the array is stored in a row-major format in the off-chip memory, each processor will make 8 off-chip requests in the access pattern depicted in Figure 2(b). The total volume of the data transferred by each processor is, therefore, 32 (and, note that only 16 of these are really useful). One simple strategy that can be adopted by an optimizing compiler is to interchange the loops. While this may help improve locality in some cases, in many loops, data dependences can prevent loop interchange (or similar loop transformations [4]).

In this example, our two step solution operates as follows. We first consider the data layout in memory (which is row-major) and allow each processor access the data using the best possible access pattern (as depicted in Figure 2(c)). Now, each processor has some data which are, originally, required by some other processor. In the next step, the processors engage in an all-to-all (intra-chip) communication (a special case of many-to-many communication), and data is redistributed accordingly. We can calculate the cost of this approach as follows. The number of off-chip memory accesses per processor is only 4 (as all 16 elements that need to be read are consecutive in the off-chip memory). In addition, the total data volume (transferred from the off-chip memory) is 16 (i.e., no unneeded data is transferred). Note that, as compared to the independent memory access scenario, here we reduce the number of off-chip accesses (per processor) by half (which can lead to significant execution cycles savings).

However, in this optimized strategy, we also need to perform inter-processor data re-distribution (i.e., the second step) so that each processor receives the data it originally wanted. In our example, each processor needs to send/receive data elements to/from the other three processors. Let us assume that performing an off-chip data transfer takes C_1 cycles and performing an on-chip communication (i.e., local memory-to-local memory transfer) takes C_2 cycles, the independent memory access strategy costs $8C_1$ (per processor) and our optimized strategy costs $4C_1 + 3C_2$ (again, per processor). Assuming C_1 is 50 cycles and C_2 is 10 cycles, our approach improves data access cost by 42.5% (170/400). This small example (with its simplistic cost view) shows that performing off-chip data accesses on behalf of other processors can actually improve the performance of the straightforward (independent) memory access strategy.

It should be noted that, depending on whether the local memory space is a software-managed local memory or hardware-managed cache, it might be necessary (in our approach) to interleave off-chip data accesses with on-chip communication. If the local memory is software-managed and is large enough to hold the entire set of elements that will be transferred from the off-chip memory, it is acceptable to first perform all off-chip accesses and, then, perform on-chip communication. On the other hand, if the on-chip local memory is not large enough to hold all the required elements, then we need to "tile" the computation. What tiling means in this context is that we first transfer (from the off-chip memory) some data elements (called "data tile") to fill the local memory, then perform

Benchmark	Brief Description	Cycles
Atr	Network IP address translator	11,095,718
SP	Computing the all-nodes shortest paths on a given graph	13,892,044
Encr	Secure digital signature generator and verifier	30,662,073
Hyper	Multiprocessor communication activity simulator	18,023,649
Wood	Color-based visual surface inspector	37,021,119
Usonic	Feature-based object estimation	46,728,681

Fig. 3. Brief Description of our benchmarks.

on-chip communication, and then perform the computation on this data tile (and write back the tile to the global memory if it is modified). After this, we transfer the next set of elements (i.e., the next data tile) and proceed the same way, and so on. It should be observed that if the on-chip memory space is implemented as a conventional cache, tiling the computation might be very important as there is no guarantee that all the data transferred from the off-chip memory will still remain in the cache at the time we move to the on-chip communication phase.

Note that, in a straightforward off-chip memory access strategy, each processor accesses its portion of an array without taking into consideration of the storage pattern of the array (i.e., how the array is laid out in memory). In contrast, in our approach, the processors first access the data using an access pattern, which is the "same" as the storage pattern of the array. This helps reduce the number of off-chip memory references. After that, the data are redistributed across processors so that each array element arrives in its final destination. In general, the proposed strategy will be useful when access pattern and storage pattern are different.

4 Experiments

4.1 Setup

We used six applications to test the success of our optimization strategy. The brief descriptions of these benchmarks are presented in Figure 3. While we have also access to pointer-based versions of these applications, in this study, we only used their array-based versions. These applications are coded in such a way that their input sizes can be set by the programmer. In this work, the total data sizes manipulated by these applications range from 661.4KB to 1.14MB.

We simulated execution of an on-chip multiprocessor with local and global memories using a custom simulator. Each processor is a simple, single-issue architecture with a pipeline of five stages. Our simulator takes an architectural description and an executable code. The architectural description indicates the number of processors, the capacities of on-chip and off-chip memories, and their relative access latencies. We refer to the relative access latencies of local, remote, and global memories as the "access ratio". For example, an access ratio of 1:10:50 indicates that a remote memory access is ten times as costly as a local memory

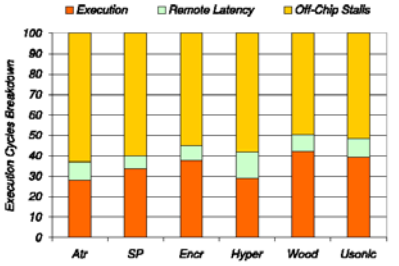

Fig. 4. Execution cycle breakdown for the default mode of execution.

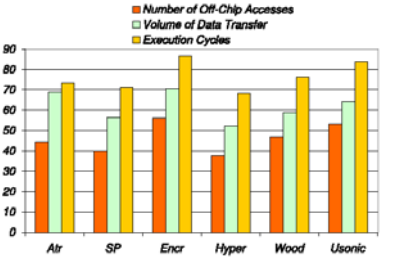

Fig. 5. Normalized number of off-chip accesses, transfer volume, and overall execution cycles.

access, and that a global memory access is five times as costly as a remote memory access. We ran the optimized code on a Sun UltraSparc machine. By default, we assume that the system has 8 processors (runnig at 250MHz) on the chip. Each processor has 4KB local memory. These processors share a 32MB global memory. The access ratio (i.e, the relative access latencies of local, remote and global memories) is 1:10:50.

In order to evaluate the effectiveness of our approach, we compare it to a "default mode of execution." In this mode, each processor performs independent off-chip memory accesses. However, before going to the off-chip memory, it first checks its local memory, and then checks all remote memories. Consequently, if the requested data item is available in either local memory or remote memories, the off-chip memory access is avoided. As discussed earlier in detail, our approach tries to improve over this default mode of execution by performing off-chip memory accesses in a coordinated fashion. In this strategy, each processor performs off-chip accesses in a way which is most preferable from the data locality viewpoint. After this, processors exchange data so that each data item arrives in its final destination. The last columns of Figure 3 gives the execution cycles obtained under this default mode of execution. Execution cycles reported in the rest of this paper are values normalized with respect to these numbers. In comparing our optimization strategy to the default mode of execution, we used three different metrics. The first metric is the number of off-chip accesses. The second metric is the transfer volume, which is the amount of data transferred from the off-chip memory. As discussed earlier in the paper, our approach is expected to transfer fewer data items (as it exploits spatial locality in off-chip accesses) than the default mode of execution. While reductions in these two metrics indicate the effectiveness of our approach, the ultimate criterion is the reduction in overall execution cycles, which is our third metric.

4.2 Results

Figure 4 gives the execution cycle breakdown when each processor accesses the global memory independently (default mode of execution). In this graph, execution cycles are divided into three parts: (1) cycles spent in executing code

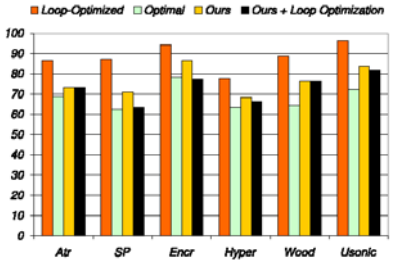

Fig. 6. Normalized execution cycles of different versions.

(including local memory accesses); (2) cycles spent in performing remote memory accesses; and (3) cycles spent in performing global memory accesses (i.e., off-chip accesses). Our first observation is that, on the average, 56.3% of cycles are expended in waiting for off-chip data to arrive. This observation clearly underlines the importance of reducing the number of off-chip data requests. In comparison, the cycles spent in remote memory accesses constitute only 8.75% of total cycles.

The graph in Figure 5 shows the number of off-chip memory accesses, transfer volume, and execution cycles resulting from our strategy, each normalized with respect to the corresponding quantity incurred when the default execution mode is used. We see that the average reductions in the number of off-chip memory accesses and transfer volume are 53.66% and 38.18%, respectively. These reductions, in turn, lead to 23.45% savings in overall execution cycles. As a result, we can conclude that our approach is much more successful than the default mode of execution. Execution cycle savings are lower in *Encr* and *Usonic* as the original off-chip access patterns exhibited by these applications are relatively good (as compared to the access pattern of the other applications under the default mode of execution).

One might argue that classical loop transformations can be used for achieving similar improvement to those provided by our optimization strategy. To check this, we measured performance of the loop optimized versions of our benchmark codes. The loop transformations used here are linear transformations (e.g., loop interchange and scaling), iteration space tiling, and loop fusion. To select the best tile size, we employed the technique proposed by Coleman and McKinley [3]. The first bar of each benchmark in Figure 6 gives the execution cycles of this version, normalized with respect to the original codes (note that both the versions use the default mode of execution). The average improvement brought by this loop-optimized version is 11.52%, which is significantly lower than the average improvement provided by our approach, which is 23.45% (the results of our approach are reproduced as the third bar in Figure 6). The reason that our approach performs better than classical loop transformations is related to data dependences [4]. Specifically, a loop transformation cannot be used if it violates the intrinsic data dependences in the loop being optimized. In many of the loops in our benchmark codes (specifically, in 45% of all loop nests), data dependences

prevent the best loop transformation from being applied. However, this result does not mean that our optimization strategy cannot take advantage of loop transformations. To evaluate the impact of loop optimizations on the effectiveness of our approach, in our next set of experiments, we measured the execution cycles of a version that combines our approach with loop optimizations. The loop optimizations employed are the ones mentioned earlier. Specifically, we first restructured the code using loop transformations such that spatial locality is improved as much as possible. We then applied our approach in performing off-chip data accesses. The last bar in Figure 6 for each application gives the normalized execution cycles for this version. One can observe that average improvements due to this enhanced version is 26.98% (a 3.53% additional improvement over our base strategy). It is also important to see how close our optimization comes to the optimal in reducing execution cycles. In order to see this, we performed experiments with a hand-optimized version that reduces the off-chip activity to a minimum. It achieves this using a global optimization strategy which considers all nests in the application together and by assuming an "infinite amount" of on-chip storage. In other words, for each data item, if possible, an off-chip reference is made only once (and since we have unlimited on-chip memory, no data item needs to be written back – even if it is modified). The second bar for each application in Figure 6 presents the behavior of this strategy. We see that the average execution cycle improvements is around 31.77%, indicating that our approach still has a large gap to close to reach the optimal.

5 Concluding Remarks

On-chip multiprocessing can be aimed at systems that require high performance in either instruction-level parallelism (ILP) or thread-level parallelism, since additional threads can be run simultaneously on other CPU cores within the chip, and therefore, is expected to have a clear advantage over traditional architectures that concentrate on ILP. In this paper, we demonstrate how a compiler can optimize off-chip data accesses in an on-chip multiprocessor based environment by allowing collective off-chip accesses, whereby each processor performs off-chip accesses on behalf of other processors. Our results clearly show the benefits of this approach.

References

1. S. P. Amarasinghe et al. Multiprocessors from a Software Perspective. *IEEE Micro Magazine,* June 1996, pages 52–61.
2. A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits.* IEEE Press, 2001.
3. S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation,* La Jolla, CA, June 1995.
4. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.