Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy

Andrea Marongiu
University of Bologna
Via Risorgimento, 2
40136 Bologna - Italy
Email: a.marongiu@unibo.it

Luca Benini
University of Bologna
Via Risorgimento, 2
40136 Bologna - Italy
Email: luca.benini@unibo.it

***Abstract*—OpenMP is a de facto standard interface of the shared address space parallel programming model. Recently, there have been many attempts to use it as a programming environment for embedded MultiProcessor Systems-On-Chip (MPSoCs). This is due both to the ease of specifying parallel execution within a sequential code with OpenMP directives, and to the lack of a standard parallel programming method on MPSoCs. However, MPSoC platforms for embedded applications often feature non-uniform, explicitly managed memory hierarchies with no hardware cache coherency as well as heterogeneous cores with heterogeneous run-time systems.**

In this paper we present an optimized implementation of the compiler and runtime support infrastructure for OpenMP programming for a non-cache-coherent distributed memory MPSoC with explicitly managed scratchpad memories (SPM). The proposed framework features specific extensions to the OpenMP programming model that leverage explicit management of the memory hierarchy. Experimental results on different real-life applications confirm the effectiveness of the optimization in terms of performance improvements.

I. Introduction

The advent of MPSoCs on the marketplace raised the necessity for standard parallel programming models. Exploiting parallelism has historically resulted in significant programmer effort, which could be alleviated by a programming model that exposes those mechanisms that are useful for the programmer to control, while hiding the details of their implementation.

OpenMP is a de-facto standard for shared memory parallel programming. It consists of a set of compiler directives, library routines and environment variables that provide a simple means to specify parallel execution within a sequential code. It was originally used as a programming paradigm for SMP machines, but in the recent past many MPSoC-suitable implementations of translators and runtime environments have been proposed [4][8][9]. OpenMP specification provides a model for parallel programming that is portable across different shared memory architectures. The interface is invariant, while different implementations can match many architectural templates, each dealing with specific hardware features. Customising an implementation to a target platform allows getting high performance on that machine. However, achieving an efficient implementation is challenging because of three main reasons. First, MPSoCs are typically heterogeneous architectures, with different processing tiles featuring non homogeneous OS support. Second, they are often characterized by a complex memory hierarchy, with a physically distributed shared memory. Finally, the implementation of synchronization directives must be carefully optimized since they strongly impact the performance of the parallelized code.

In this paper we present an OpenMP implementation for a non-cache-coherent MPSoC with explicitly managed memory hierarchy. Code transformation and calls to the runtime are automatically instantiated by a customized GCC 4.3.2 OpenMP compiler. We address all the main issues concerning support of parallel execution in the absence of a homogeneous threading model, data sharing between processors and efficient implementation of the synchronization (`barrier`) directive.

A key element in achieving an efficient OpenMP implementation for MPSoCs is exploitation of the local memories. OpenMP does not provide very useful means to exploit data locality, nor to cope with physically distributed shared memory. With the proposed framework we make the following contributions:

1) We present specific extensions to the OpenMP model that allow the programmer to specify with custom directives and clauses which arrays are likely to achieve significant benefits when mapped to local memories
2) We provide an optimized implementation of the proposed interface
3) We couple the provided custom compiler support with tools for automated array access profiling and efficient allocation

Specifically, the accesses to the candidate arrays annotated with custom `#pragma`s are monitored during an off-line profiling phase to determine which data items are more frequently referred within a given array and by which processor. These data are allocated by the custom runtime support onto the local scratchpads (SPM). When entire arrays do not fit into SPMs, the memory allocator enables a second compilation step. Every reference to target arrays is instrumented in order to partition it in chunks that will be mapped to different SPMs based on access count. Experimental results show that this optimization allows significant improvements on many benchmarks with repect to the use of a data cache for the considered MPSOC architecture.

II. Related Work

OpenMP implementations for MPSoCs have been presented in [4][8][9]. In [4] the authors deal with implementing parallel tasks in the absence of OS. They provide optimized implementation of the `barrier` directive and `reduction` clause, but they do not provide any optimizations aimed at exploiting local memories. In [8][9] custom extensions to the OpenMP API are given to enable parallel execution on DSPs. They also provide custom directives to optimize memory allocation exploiting scratchpads. Unlike ours, their approach is dynamic, in that it relies on data movement through DMA, but it is subject to transfer latency. To alleviate it the authors propose data pre-fetching and double buffering optimizations. The authors of [12] present dataflow analysis techniques that take into account OpenMP semantics and provide a series of optimizations such as removal of unnecessary cache flushes to preserve coherence, redundant barrier removal and privatization of dynamically allocated objects. Compiler support for explicitly managed memory hierarchies

978-3-9810801-5-5/DATE09 © 2009 EDAA

has been proposed in [7]. Here the authors propose an intermediate representation which model programs as bulk operations (data transfers or kernels). Such level of abstraction allows them to efficiently partition data across the different levels of the memory hierarchy.

Static methods for efficient mapping of data items to SPMs are discussed in [2][11][13]. In [2] is proposed a compiler-directed method for allocating heap data onto SPMs. In [11][13] the use of SPM is proposed as an alternative to caches on MPSoCs. The authors of [11] present a technique for partitioning the application's scalar and array variables into off-chip DRAM and on-chip SPM with the goal of minimizing the total execution time of embedded applications. In [13] a compiler-based technique is proposed to analyze the program and select both program and data parts to be places into the SPM. They compare energy results with an equivalent cache-based solution. Although similar to our static allocation approach, all these works don't focus on data partitioning on multiple scratchpads.

Dynamic allocation techniques are proposed in [14]. Here the authors partition the program into regions and allow changes in the allocation at the beginning of each region (allocation is fixed within regions). Most frequently used data in that region are copied into the scratchpad. Dynamic approaches are also presented in [5][6], where the authors propose a compiler strategy aimed at reducing extra off-chip memory accesses caused by interprocessor communication. This optimization exploits reuse of data in SPM, and is based on loop tiling and on the optimal choice of the tile processing schedule that minimizes interprocessor communication. Although these dynamic approaches have the advantage of reusing the SPM during program execution, they focus on loop nests and need precise information about the loop structure and array access patterns to guarantee the legality of the tiling transformation. Our profile-based approach can deal with a broader range of parallel constructs, and does not require precise information on memory access pattern at compile time.

III. Target Architecture

Figure 1 shows the simplified block diagram of the target architectural template. It consists of a configurable number of processing tiles, each of which features private L1 data and instruction caches, as well as a local scratchpad. Each processor has access to on-chip private and shared memory, as well as a big off-chip shared DRAM memory. Support for synchronization is provided through a special hardware semaphore device.

This architectural template matches several existing MPSoC designs such as the IBM Cell BE, TI OMAP, Cradle 3SoC, in which a high performance CPU is coupled with an array of DSPs or generic hardware accelerators (e.g. Image Processing Units, video/audio accelerators). These coprocessors usually run custom lightweight runtime middleware rather than homogeneous OSes, they typically don't use MMUs and hardware cache coherency is not supported. To guarantee a consistent view of the shared memory from concurrent multiprocessor accesses it can be either configured to be non-cacheable or explicitly managed with software-controlled cache flush operations.

These features call for specific modifications to the standard OpenMP interface and implementation, as discussed in the following Sections IV and V.

IV. OpenMP Support Framework

An OpenMP implementation consists of a code translator and a runtime support library. The framework presented in this paper is based on the GCC 4.3.2 compiler, and its OpenMP translator (GOMP). The OMP pragma processing code in the parser and the

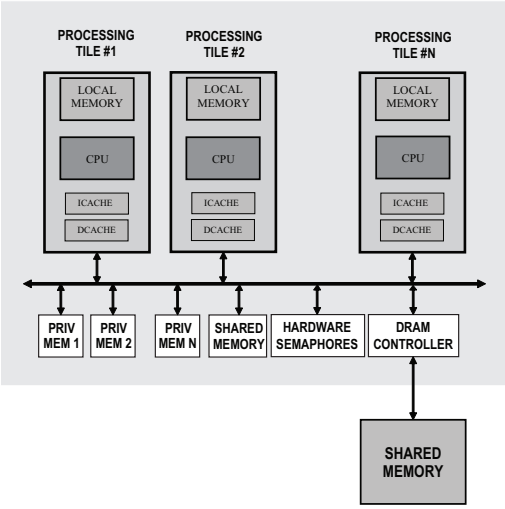


Fig. 1. Shared memory architecture.

lowering passes have been customized to match the peculiarities of our architecture. All the major directives and clauses are supported by our implementation. Work-sharing constructs support all kinds of static and dynamic scheduling, and all synchronization constructs are supported. The runtime library (libgomp) has been re-implemented from scratch due to the absence of an homogeneous threading model. The original implementation of the libgomp library is designed as a wrapper around the *pthreads* library. A `#pragma omp parallel` directive is handled outlining the code within its scope into a new function that will be mapped to parallel threads through a call to `pthread_create`.

In our case we need to take a different approach, since we cannot rely on a thread library that allows us to fork a thread onto another core. Our runtime library implements the `main()` function. Each processor loads its executable image of the program, and then the execution is differentiated between master and slave threads based on the core ids. After an initialization step, executed by every thread, the master calls the application `main()` function while the slaves wait on a barrier. Our custom translator replaces every `#pragma omp parallel` directive with a call to our library function `do_parallel()`, and passes it two arguments: the address of the parallel function code, and the address of a memory location where shared data is placed. In the function `do_parallel()` the master updates the addresses at which the slaves will look for the parallel function and the shared data. At this point it releases the barrier and all the threads can execute concurrently the outlined function. At the end of the parallel region the master core continues executing sequential parts of the application, while the slaves come back on the barrier.

In the OpenMP programming model barriers are often implied at the end of parallel regions or work-sharing directives. For this reason they are likely to overwhelm the benefits of parallelization if they are not carefully designed taking into account hardware peculiarities and potential bottlenecks. In our previous work [10] we tested the scalability of different barrier implementations, and we noticed that with an architectural template consisting in 8 cores and a shared bus interconnect, there is a serious performance penalty when many cores are idling on the barrier. This may happen when the application shows load imbalance in a parallel region, or when most of the execution time is spent in serial regions. The latter happens because the remote

polling activity of the slaves on the barrier fills the interconnect with access requests to the shared memory and to the semaphore device. To address this issue we devised a barrier implementation that exploits the scratchpad memories. The master core keeps an array of flags representing the status of each slave (entered or not), and notifies on each slave's local memory when everybody has arrived on the barrier. Moving the polling activity from the shared memory to local SPM and eliminating the necessity for lock acquisition allows us to remove the congestion on the interconnect.

V. Profile-Based Efficient Data Allocation

Efficient utilization of the on-chip memory is of the utmost importance to exploit the computational power of MPSoCs. The advantages in placing data very close to the processor are significant in terms of both energy and access latency. The cache memory is clearly the most obvious implementation of this idea, but it may not be the most efficient solution on MPSoCs. It is not uncommon that caches are not used at all in MPSoC designs (e.g. the Cell BE). The natural alternative to a data cache on such systems is the scratchpad memory. Scratchpads are less energy consuming [1] and more predictable than data caches. Though their access latencies are equal (typically 1 cycle), SPM always guarantee the same access time, whereas an access to cache is subject to conflict misses, thus making it very difficult to predict execution time accurately. Furthermore, on our target architecture cache coherency is not hardware-supported. To mantain a consistent view of the memory between the processors there is the need to perform a flush of the shared data at each synchronization point. This rationale motivated the idea of providing, within the OpenMP programming model, a means for the programmer to specify the arrays on which most of the elaboration is done, and that consequently should allow significant performance improvements if mapped to local SPM.

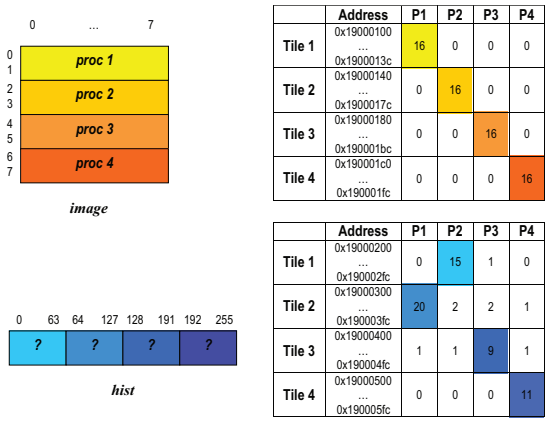
OpenMP programs are heavily focused on parallel loops, and the most common parallelization scheme is that in which the iteration space is partitioned among threads in contiguous chunks. When data (array) space and iteration space overlap, different threads access exclusively different data slices (hereafter called tiles). This happens with the *image* array in the following example code for histogram creation:

```
#pragma omp parallel shared(hist,image) num_threads(4)
#pragma omp for
  for (i=0; i<8; i++)
    for (j=0; j<8; j++)
      hist[image[i][j]]++;
```

The iteration space of the outer loop is split in four chunks, assigned to different threads. Since the *image* matrix is indexed with the loop induction variables, each processor will access a different tile. In this case, static compiler analysis is sufficient to devise an array partitioning strategy that allocate tiles onto the SPM of the processor that exclusively accesses them. This is illustrated in fig. 2 (same shades of yellow for the array tile and the processor SPM onto which it is mapped).

On the other hand is impossible to foretell the access pattern for the *hist* array at compile time (blue tiles in fig. 2). A possible solution to deal with array partitioning in this case is that of creating as many tiles as processors – like in the previous case – and relying on application profiling [3] to map each tile onto the SPM of the processor that accesses it more frequently.

Partitioning an array and mapping the segments onto different physical memories (i.e. addresses) requires address translation at each array access. In the absence of harware support for that



	Address	P1	P2	P3	P4
Tile 1	0x19000100 ... 0x1900013c	16	0	0	0
Tile 2	0x19000140 ... 0x1900017c	0	16	0	0
Tile 3	0x19000180 ... 0x190001bc	0	0	16	0
Tile 4	0x190001c0 ... 0x190001fc	0	0	0	16

	Address	P1	P2	P3	P4
Tile 1	0x19000200 ... 0x190002fc	0	15	1	0
Tile 2	0x19000300 ... 0x190003fc	20	2	2	1
Tile 3	0x19000400 ... 0x190004fc	1	1	9	1
Tile 4	0x19000500 ... 0x190005fc	0	0	0	11

Fig. 2. Motivation example for profile-based array partitioning. Each array partition is allocated to the scratchpad of the processor that accessed it most frequently

purpose this task has to be accomplished in software. Since manually instrumenting memory accesses in a program is a tedious and error-prone task (in contrast with the ease of use of the OpenMP API), we seamlessly integrate this process in the compiler, making it invisible to the programmer who simply has to annotate candidate arrays with custom OpenMP directives and clauses.

The compiler-generated instructions for address translation rely on metadata associated to the array, containing the base address at which each tile can be found. These addresses can be provided by the programmer, but this would require precise knowledge of the program behavior and array access pattern. To deal with this problem, orthogonally to the code instrumentation task, we developed tools to automatically find the most efficient mapping of array portions to scratchpads.

We enhanced our simulator to monitor every array access from every processor during a first program run. The output of this step is a trace containing the addresses of each array location, and the associated access count (for each processor). Based on the access frequency information we allocate array tiles onto the SPM of the processor that accessed it most frequently. This is done by automatically filling in the metadata associated to distributed arrays. This information is loaded by our custom OpenMP runtime during a second program run, and it is exploited to compute array address translation. The complete tool-flow is depicted in fig. 3.

A. OpenMP Extensions

Candidate array variables can be declared within the code with the custom **`distributed`** directive

```
int a[10][10];
double b[100];
#pragma omp distributed(a,b)
```

The semantics of this directive is affected by the possible coupling with one of two custom clauses, **`tiled`** and **`split`**. These are provided to cope with three different scenarios.

1) If entire arrays fit into a SPM no tiling should take place. This is clearly the most efficient solution, since once the base address of an array is known to all processors, they can simply access it preserving the original pattern. Consequently there is no need for further complex code transformations, and no performance

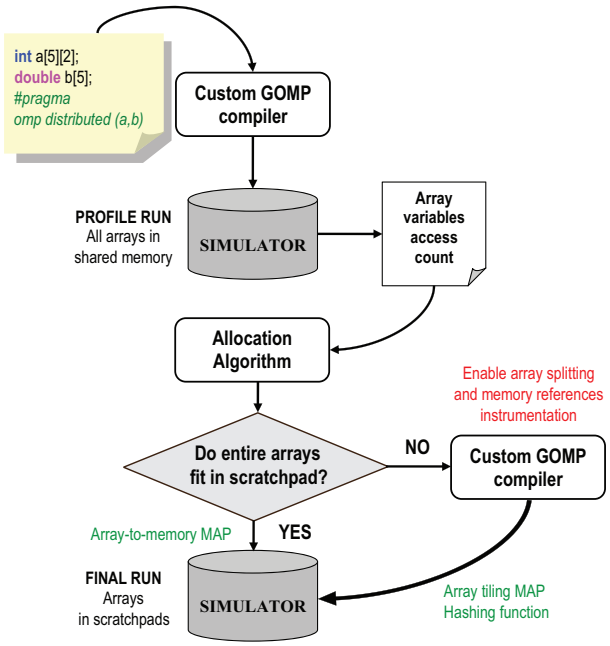


Fig. 3. Execution flow of the optimizer.

penalties due to the overhead such transformations bring. Using the **distributed** directive with no associated clause informs the compiler about a similar scenario. To explicitly map different arrays throughout the global address space, the compiler transforms the static array declaration into a pointer. and instantiates a call to the **distributed_malloc()** runtime function for each variable declared within the scope of a **distributed** pragma.

```
int **a;
double *;
...
a = (int *)distributed_malloc(10*10*sizeof(int));
b = (double *)distributed_malloc(100*sizeof(double));
```

This function is a custom **malloc()** that relies on array access information collected during a profiling run of the program. If no information is available, data is simply allocated in the off-chip shared memory, otherwise arrays are allocated according to the following rules:

- Arrays with overall maximum access frequency are allocated first.
- Each array is allocated onto the SPM local to the processor that accessed it most frequently.
- If there is not enough space on a SPM, the optimizer allocates the array on a remote SPM.

2) When an entire array does not fit into a SPM tiling is mandatory. In the (worst) general case – when multiple processors may access different tiles – for every array reference there is the need to check which tile is currently being accessed and at which offset. This happens with the *hist* array in fig. 2, and brings the highest overhead. When a **distributed** array is declared with the **tiled** clause, it is partitioned into a number of tiles equal to the number of cores (threads) participating in a parallel region. This implies the need to instrument every array access in order to guarantee that the correct memory location is accessed by each thread. Every statement containing a distributed array access

```
/* Original code */
a[i*10+2][j] = foo();
```

is replaced with another statement referring to a new address computed by means of some overhead instructions

```
/* Overhead instructions */
/* Compute tile ID which the memory reference belongs to */
int tile = ...;
/* Lookup base address for current array tile */
int *base = tiles[tile] ;
/* Compute index for a location within the tile */
int index = ... ;

/* Transformed memory access */
base[index] = foo();
```

The SPM address that replaces the original memory reference is obtained by looking at the *tiles* metadata matrix which contains the base address for each tile of every array declared as **distributed**. The code to populate this matrix is generated at the end of the memory allocation stage. It is loaded by the runtime environment during execution, and made visible to the main program.

3) As already discussed, often arrays in OpenMP programs are accessed in such a way that each processor operate on a different tile, without sharing any location. Profiling information are useful when this access pattern is statically undetectable – either in case of input-dependent access pattern, or conservative compiler assumptions. The compiler can be made aware of this pattern through the use of the **split** clause. In this case array references are instrumented in such a way that there is no check of which tiles is being accessed, thus reducing the address translation complexity and its overhead.

Since the access pattern to an array may change at different parallel regions in the program, these clauses can be coupled with the use of the **parallel** directive, and with the worksharing directives **for** and **sections**. The code excerpt below gives and example of the use of these clauses on the histogram example.

```
#pragma omp parallel for tiled(hist) split(image)
  for (i=0; i<8; i++)
    for (j=0; j<8; j++)
      hist[image[i][j]]++;
```

VI. Experimental Results

In this Section we describe our experimental setup and provide performance results achieved on a set of code kernels and real applications.

A. Virtual Platform Configuration

processors	4 RISC, 200Mhz
D-cache	16KB, 4 way set assoc., write-through lat 1 cycle
I-cache	8KB, direct mapped lat 1 cycle
private memory	lat 2 cycles
on-chip shared memory	lat 2 cycles
off-chip shared memory	lat 70 cycles
AMBA AHB	32 bit, 200Mhz, arb 2 cycles

TABLE I
VIRTUAL PLATFORM CONFIGURATION PARAMETERS

Table I sumarizes the configuration parameters employed for our experiments. As we already pointed out in Section III hardware cache coherency is not supported. To guarantee a consistent view of the shared memory from concurrent multiprocessor accesses it can be configured to be non-cacheable but in this case it can only be inefficiently accessed by means of single transfers. Cacheability of the shared memory can be toggled, but in this case explicit software-controlled cache flush operations are needed.

B. Benchmarks and Setup

To test our OpenMP programming framework and the proposed custom optimizations we use the four microbenchmarks described in fig. 4. For each application, we measure the execution time of various

BENCHMARK	SOURCE	DESCRIPTION
Matrix multiplication	Fox Algorithm	Dense linear algebra kernel
Loop with dependencies	OmpSCR (OpenMP Source Code Repository)	A series of loop nests with loop carried dependencies.
LU Decomposition	NAS PB	Dense linear algebra kernel. Lower and upper triangular systems.
IS (Integer sort)	NAS PB	Multiple levels of memory reference indirection.

Fig. 4. Benchmarks

program runs enabling one of the following optimizations:

- Cacheable - The shared memory is cacheable, but since no hardware support is given for cache coherency the program outcomes are NOT CORRECT. This configuration is used to estimate the results achievable with the use of coherent caches. Clearly these results are to be considered an upper bound, since we are not taking into account the cost for invalidating stale cache lines.
- Flushes - The shared memory is cacheable. The runtime system and the benchmarks are instrumented to explicitly flush shared data from the caches at each synchronization point.
- Distributed - The shared memory is NOT cacheable. Arrays declared as having distributed allocation semantics with our custom directive are instantiated in the SPMs. No array splitting takes place.
- Tiled - The shared memory is NOT cacheable. Entire arrays DO NOT fit into scratchpads, so array partitioning takes place. Different tiles of the same array are instantiated onto (possibly) different scratchpads according to the access count profile.
- Split - The shared memory is NOT cacheable. The memory allocator has detected that no overlapping accesses from different processors take place. The parallel regions are annotated with the custom **`split`** clause, and most of the overhead instructions are removed.

The speedups are referred to the execution time of the configuration with no optimizations, namely with arrays allocated in the off-chip shared memory, which is not cacheable.

C. Discussion

Matrix Multiplication - Most of the computation in this benchmark is done within a single parallel region, but two barriers are required to synchronize sub-matrices multiplication between different cores and to prevent the slaves to run the following loop iteration while the master updates the whole matrix. This

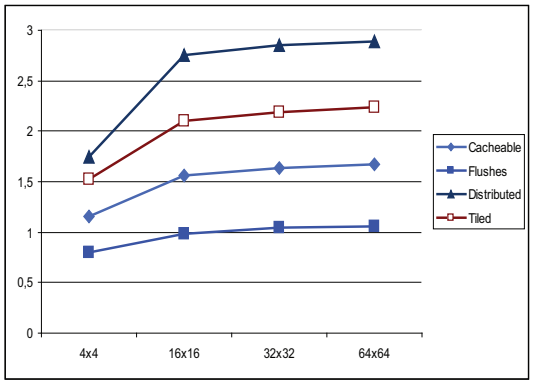


Fig. 5. Speedup for the matrix multiplication benchmark. On the x-axis is represented the size of the matrix (number of integer elements).

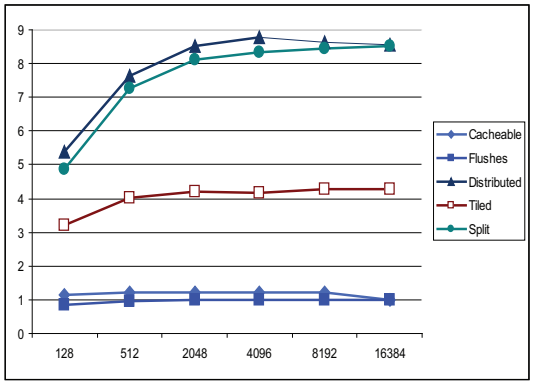


Fig. 6. Speedup for the loop with dependencies benchmark. On the x-axis is represented the size of the arrays (number of integer elements).

requires two flush instructions per iteration, which carry a significant overhead, as shown in figure 5. When the size of the matrix is small (up to 16x16 integers) the short execution time of the program causes the cost for these flushes to be predominant, thus leading to execution slowdown. For matrix sizes over 32x32 the speedup achievable with the (ideal) coherent caches – roughly 1.6X – is reduced to 1.05X in presence of the software flushes. The *distributed* curve show the greatest speedup, as we expected, since data is always on-chip and no penalty has to be paid for data movement or overhead instructions. The *tiled* curve show very good speedup results (2.2X) for the array partitioning, better than those achievable with coherent caches, that are limited by the poor data reuse shown by the benchmark.

Loop With Dependencies - This benchmark features two parallel regions with threads accessing different non-overlapping portions of two arrays annotated with the **`distributed`** pragma. The allocator detects the access pattern and enables array partitioning with the **`split`** clause to remove unnecessary overhead instructions. In figure 6 we report the results for different sizes of the array. It has to be pointed out that 4096 is the limit size for the arrays to fit in SPM, so for bigger sizes array partitioning is mandatory. As already pointed out for the matrix multiplication, since the execution time of the code kernel is very short (it basically performs array population, without any computation), the cost for the software flushes is not negligible, thus limiting any speedups in the *flushes* curve. When the size of the array is small (hundreds of elements)

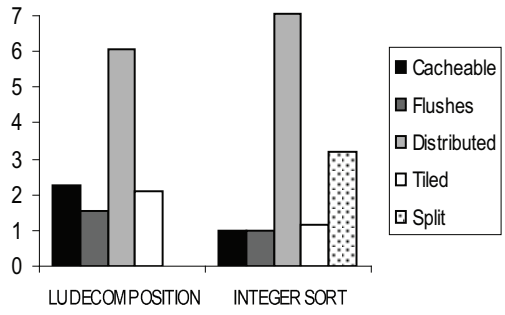


Fig. 7. Speedup for the LU and IS benchmarks

this cost overwhelms the benefits introduced on memory latency by the cache, thus leading to slowdowns. As the size of the arrays grow larger this cost is amortized, but since there is no data-reuse in the application, even using ideal coherent caches does not bring any significant speedup. On the contrary, due to the memory-bounded nature of the application, having entire arrays on the local SPM allows up to 8.6X speedups. Array tiling still brings significant benefits (4.2X) notwithstanding the overhead instructions, and since this benchmark is the perfect example for our custom **`split`** clause, the speedup can be boosted up to 8.4X. Since a high percentage of the parallelism in OpenMP programs is found in doall loops of this kind (initialization and population kernels), this finding is representative and generalizable to many similar applications.

LU Decomposition - The core of the computation takes place inside a parallel region forked within a loop nest. This calls for repeated invocations to the runtime for data flushes, which significantly impact performance, as can be seen comparing the *cacheable* and *flushes* bars in figure 7. Array partitioning allows 2.1X speedup, slightly less than the 2.2X achieved with the coherent cache, but way more than the 1.55X of the cache with flushes. The big difference with the *distributed* bar is justified by the fact that several different locations of the arrays are referenced within three-level nested loops. Since different processors access the entire arrays, every memory reference require all of the overhead instructions for address computation. No ***split*** clause can be applied to the parallel regions of the benchmark.

IS (Integer Sort) - None of the three main arrays of the program fit in one SPM, so array partitioning is mandatory. We provide results for the *distributed* optimization as a reference of what is achieved keeping entire arrays in a bigger SPM. Since this application is CPU-intensive the cost for flushes is negligible. The key ranking kernel shows a very complex memory access pattern, as arrays are often indexed with subscripted subscripts. This limits the benefits of the cache – due to the misses – as well as the tiling, which requires a big amount of instrumentation. Since several locations are referenced within a single iteration the array partitioning only provides a 1.15X speedup. On the other hand, this code kernel is a candidate for our ***split*** clause. Removing the unnecessary overhead instructions brings the speedup up to 3.2X.

VII. Summary and Conclusions

In this paper we presented a custom OpenMP implementation suitable for non-cache-coherent MPSoCs with explicitly managed memory hierarchy. Custom directives and clauses are coupled with a profiler and a memory allocator we designed to automate the process of mapping array portions onto the scratchpad of the processor that accessed those memory locations more frequently during a profile run. Experimental results on a set of four representative benchmarks and code kernels confirm the effectiveness of the proposed optimizations. Future work will be focused on exploiting per-parallel region (rather than program-wide) profile information. Coupling this information with compiler support to dynamically move data throughout the memory hierarchy promises better performance. Also, compiler and runtime optimizations for address translation can be devised to reduce the instrumentation overhead.

Acknowledgment

This work was supported by the SHARE Project funded by the European Community, Contract FP7-ICT-224170.

References

[1] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design & Test of Computers, IEEE*, 17(2):74–85, Apr-Jun 2000.

[2] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, 2005.

[3] A. C. Federico Angiolini, Luca Benini. An efficient profile-based algorithm for scratchpad memory partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(11):1660–1676, 2005.

[4] W.-C. Jeun and S. Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 44–49, 23-26 Jan. 2007.

[5] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(3):281–287, March 2004.

[6] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2):243–260, Feb. 2004.

[7] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 226–236, 2007.

[8] F. Liu and V. Chaudhary. Extending openmp for heterogeneous chip multiprocessors. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 161–168, 6-9 Oct. 2003.

[9] F. Liu and V. Chaudhary. A practical openmp compiler for system on chips. *International Workshop on OpenMP Applications and Tools, WOMPAT 2003*, pages 54–68, June 2003.

[10] A. Marongiu, L. Benini, and M. Kandemir. Lightweight barrier-based parallelization support for non-cache-coherent mpsoc platforms. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 145–149, New York, NY, USA, 2007. ACM.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, 2000.

[12] S. Satoh, K. Kusano, and M. Sato. Compiler optimization techniques for openmp programs. In *Scientific Programming*, pages 9–2, 2001.

[13] S. Steinke, L. Wehmeyer, B. sik Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. *In Proceedings of the conference on Design, automation and test in Europe*, page 409, 2002.

[14] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[15] www.openmp.org. Openmp application program interface v.3.0.