Lightweight Barrier-Based Parallelization Support for Non-Cache-Coherent MPSoC Platforms

Andrea Marongiu
DEIS – University of Bologna
Viale Risorgimento 2
40133 Bologna
amarongiu@deis.unibo.it

Luca Benini
DEIS – University of Bologna
Viale Risorgimento 2
40133 Bologna
lbenini@deis.unibo.it

Mahmut Kandemir
Dept. of Comp. Sc. and Eng.
Penn State University
University Park, PA 16802
kandemir@cse.psu.edu

ABSTRACT

Many MPSoC applications are loop-intensive and amenable to automatic parallelization with suitable compiler support. One of the key components of any compiler-parallelized code is barrier instructions which are used to perform global synchronization across parallel processors. This scenario calls for a lightweight synchronization infrastructure.

In this work we describe a lightweight barrier support library for a non-cache-coherent MPSoC architecture. The library is coupled with a parallelizing compiler front-end to set up a complete automated flow which, starting from a sequential code, produces the parallelized binary code that can be directly executed onto an MPSoC target (a multi-core non-cache-coherent ARM7 platform). This tool-flow has been characterized in terms of system performance and energy.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Run-time Environments*

General Terms

Performance

Keywords

Barrier synchronization, code parallelization, MPSoCs

1. INTRODUCTION

MPSoCs can execute multiple instruction streams in parallel, thereby achieving coarse grain parallelism (thread level parallelism). In addition, many embedded applications have multiple distinct components (modules) that could be best optimized if mapped to customized heterogeneous cores.

One of the key problems to be addressed in order to harness potential computational power of these parallel systems is *code parallelization,* which can be described as decomposing the application code into parallel threads and assigning these threads to parallel cores for execution. While a serial application can be parallelized by a knowledgeable programmer, this is in general not a trivial task. Therefore, compiler support for automated code parallelization can be very useful in practice.

Many data-intensive embedded applications are loop intensive and are amenable to automatic parallelization with suitable compiler support. One of the key components of any compiler-parallelized code is *barrier instructions* which are used to perform global synchronization across parallel processors. As compared to programmer-parallelized codes, compiler-parallelized codes can contain larger number of barriers, mainly because a compiler has to be conservative in parallelizing an application (to preserve the original sequential semantics of the program), and this means, in most cases, inserting extra barrier instructions in the code. Also, apart from the performance overheads they bring, barriers cause significant power consumption as well (as evidenced by recent research [8]), and this power cost increases as the number of cores is increased.

Motivated by these observations, this paper makes the following contributions:
1) It describes an MPSoC-suitable lightweight implementation of the runtime synchronization facilities used by a parallelizing compiler frontend, with particular emphasis on barrier implementation. In order to avoid overheads due to multiple software layers the approach does not require OS support. This runtime library was coupled with the optimizing compiler to obtain a fully automated tool flow.
2) It presents experimental results which give a detailed cost analysis of loop parallelization, when using the proposed barrier implementation.
3) It discusses uses of this barrier construct and explains how they may affect parallelization decisions taken by a compiler.

2. RELATED WORK

Different schemes have been proposed for loop parallelization within different domains. In the context of high-end computing, fundamental relevant studies include [1, 2, 3, 4].

Xue et al [5] explore a resource partitioning scheme for parallel applications in an MPSoC. Ozturk et al [6] propose a constraint network based approach to code parallelization for embedded MPSoCs, and Lee et al [7] present a core mapping algorithm that addresses the problem of placing and routing the operations of a loop body.

On the side of barrier synchronization many related work propose hybrid hardware-software approaches to achieve both fast synchronization and power savings. Liu et al [8] discuss an integrated hw/sw barrier mechanism that tracks the idle times spent by a processor waiting for other processors to get to the same point in the program. Using this knowledge they scale the frequency of the cores thus achieving power savings without compromising the performances. Sampson et al [9, 10] present a mechanism for barrier synchronization on CMPs based on cache-lines invalidation. They ensure that all threads arriving at a barrier require an unavailable cache line to proceed, and, by placing additional hardware in the shared portions of the memory subsystem, they starve

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

their requests until they all have arrived. Li and al [11] present a mixed hw/sw barrier mechanism to saving energy in parallel applications that exhibit barrier synchronization imbalance. Their approach transitions the processors arriving early at the barrier to a low power state, and wake them back up when the last processor gets there. Useful surveys on synchronization algorithms for Shared Memory Multiprocessors were the works of Kumar et al [12] and Mellor-Crummey [13].

3. TARGET ARCHITECTURE

Our target MPSoC architectural template (depicted in Figure 1) consists of a configurable number of RISC cores, each of which has its own cache. Moreover, all cores can access a shared memory device to which the shared addressing space is mapped, and a special hardware device dedicated to synchronization. This device provides the *test-and-set* feature – needed to ensure the atomicity of a lock-acquiring operation – on which we model our implementation of synchronization facilities. This synchronization hardware can be viewed as a special memory, mapped within the address space of cores. Reading from one location in this memory has however a different semantic: if the content of a location is zero, a read operation returns the zero value and atomically stores a one into it.

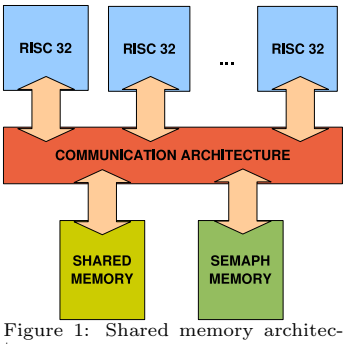

Figure 1: Shared memory architecture.

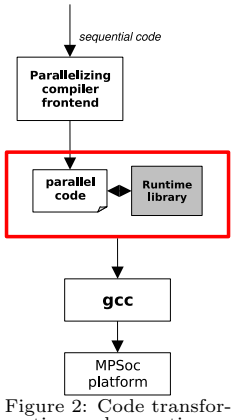

Figure 2: Code transformation and execution.

In this architectural template, cache coherency is not hardware supported. To guarantee data coherence from concurrent multiprocessor accesses shared memory 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.

4. SOFTWARE INFRASTRUCTURE

Figure 2 depicts our software framework. The application input code is a sequential C program. As a first step it is transformed into a parallelized code by the compiler frontend. This implementation relies on the synchronization features provided by our runtime library which is compiled together with the application code and fed to our simulator. Figure 3 describes how a serial code is transformed by the optimization pass into a parallel routine inside the parallel output code and how this interacts with the runtime library.

4.1 Parallelizing Compiler Front-End

Most of data-intensive embedded applications are loop based, i.e., they are structured as a series of loop nests, each operating on large datasets. Consequently, the most natural way of parallelizing such an application is to distribute loop iterations across parallel processors. This process is carried out by taking into account data dependencies. To parallelize a loop nest, our compiler front-end first extracts data

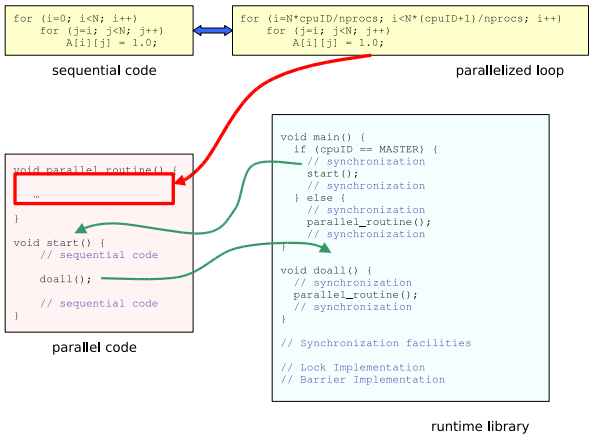

Figure 3: Original serial code and transformed parallel code. Interaction between parallel code and runtime library.

dependencies; each data dependence can be represented using a vector, called the data dependence vector. When all data dependence vectors are considered together, the compiler figures out (conservatively) which loops in the nest can be executed in parallel. Specifically, a loop can be run parallel if it does not carry any data dependency.

An important point in loop parallelization in shared memory systems is that, after each parallel loop, the compiler automatically inserts a global synchronization call (typically in form of a barrier instruction) in the code. The purpose of this instruction is to prevent any processor to get ahead of the other processors and start executing the next piece of code in execution. Note that if processors do not synchronize with each other and start executing the next piece of code together this may violate data dependences and ultimately change the original semantics of the application.

In our compiler, once available parallelism in a loop nest is identified (based on dependences), the body of the parallel loops are extracted into a separate procedure, and replaced by a call to the "doall" function (from the runtime library). The runtime library calls the procedure on each processor, and executes a barrier before returning.

4.2 Runtime Library

The support library orchestrates parallel execution synchronizing code execution on the different cores. One of our main goals was that of keeping our implementation layer as thin as possible, so the approach proposed is OS-less. We used a static task mapping, where each task is mapped on one processor identified by a unique ID.

The library also provides a set of synchronization facilities – namely locks and barriers – which implementation relies on the hardware semaphore device described in section 3. At init time all locks and barrier lock fields will be explicitly instantiated in semaphore region. This will allow atomic reading/writing from/to these variables. Counter fields of barriers will be instantiated in shared memory so that each processor can have a coherent view of their values. Since our focus in on a NCC architecture the region in which these synchronization variables are stored is never cached[1].

The parallel code implements a *start* routine which is called by the runtime library after initializations have been done. This routine contains the sequential code to be performed by the master core only, then parallel execution is initiated calling the library's *doall* method. The parallel routines – which are implemented in the parallelized code as well– are linked to the library by means of a global function pointer.

[1]This in order to preserve coherency (data consistency)

4.3 Barrier Implementation

A typical barrier implementation includes a shared counter – increased each time a task enters the barrier – and a release flag on which each worker spins. The last one which enters the barrier sets this flag and releases everybody. This implementation relies on hardware/ISA feature that grants an atomic write for lock acquisition.

Since our lock implementation relies on a special hardware accessible by the cores via the interconnect fabric, every operation on a shared variable that requires exclusive access to that resource introduces extra traffic on the bus.

As already discussed a compiler-generated parallel code includes a big number of barriers, so is beneficial to eliminate the overhead due to this additional traffic. To achieve this goal we devise a special Master-Slave form of barrier, in which the master is responsible for gathering slaves at the barrier. The release phase has been implemented in a separate function, so that the master can do some further operations before releasing the slaves. We define a barrier structure like the following:

```
typedef struct Barrier {
  int entered[NSLAVES];
  int usecount;
} Barrier;
```

The int vector *entered* is the array of flags, one per each slave, that are polled by the master to check the presence of the slaves on the barrier[2]. The field *usecount* keeps the usage count of the barrier, so it is incremented every time the master releases the barrier. It also and mostly serves as a waiting condition for the slaves, as they keep spinning until this value is increased by the master when releasing the barrier. Before entering the parallel region each slave core has to enter the barrier through a call to the following function:

```
void Slave_Enter(Barrier *b, int id) {
  int ent = b->usecount;
  b->entered[id] = 1;
  while (ent == b->usecount);
}
```

When a slave enters the barrier notifies its presence by storing a 1 in the *entered* vector at the location corresponding to its *id*. It reads the value of the variable *usecount* at entering time and busy waits until this value is changed by the master. To start a synchronization operation the master has first to call a wait function, whose implementation is shown below:

```
void Master_Wait(Barrier *b, int nprocs) {
  int i;
  for (i = 1; i < nprocs; i++)
    while (!b->entered[i]);
  // Reset flags to 0
}
```

The master scans the *entered* vector, stopping at each slave flag until it detects its presence. When all slaves reach the barrier the flags are reset to 0. At this point to release the slaves the master has to call a release function shown below:

```
void Master_Release(Barrier *b) {
  b->usecount++;
}
```

Figure 4 compares the cost[3] of a barrier invocation when using our implementation and a typical shared counters-based one. Eliminating bus traffic due to accesses to the semaphore device results in a much fewer number of cycles taken by our version. Even if in absolute terms the cost of both is not very high our version would make the difference in benchmarks with few computation and containing a large number of barrier invocations.

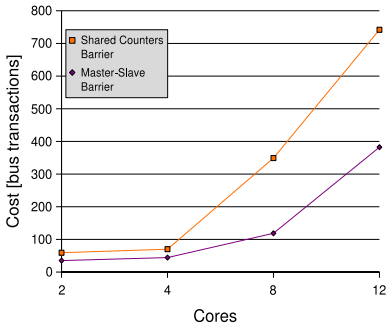

Figure 4: Comparison between our version of the barrier and a typical shared counters implementation

5. EXPERIMENTAL EVALUATION

Our automatic parallelization framework has been analyzed in detail by means of a cycle-accurate virtual platform[15] that models all essential system components. It is important to point out that our emphasis in this work is not on the effectiveness of the parallelizing compiler in discovering parallelization opportunities on benchmark programs[4]. Our main focus here is on evaluating the efficiency and scalability of our parallelization support library. Consequently, instead of conventional applications, we used synthetic benchmarks. All the simulations we made sweep both in the number of processors and in the size of data. Data are collected in two kind of plots: performance and energy.

Performance plots show the overall execution time of the parallel code (on different processor counts) compared to that of the sequential code. Execution time is partitioned into three main contributions: (i) init time, (ii) synchronization time, and (iii) parallel execution time. Init time is the time required for initialization library routines to run[5]. Synchronization time is the sum of the time spent by the cores waiting on each barrier. Effective execution time is the actual time spent over parallel computation. Each of these contributions was measured in an ideal case:

a) Synchronization time grows unpredictably with the number of cores because of the increased bus traffic, but we only want to measure the time increase due to the greater number of polled slave flags in the `for` loop. To achieve this goal, we wrote a separate small benchmark in which the master core issues the barrier only after all slave have already entered it. In this way, we can be sure the master is the only processor accessing the bus to check the shared flags, and we are sure it will only check once for each of them.

b) Ideal execution time is estimated simulating on a single processor the computational load it would have if it was running in parallel with other n cores[6]. The difference between actual timings collected from the benchmark and the ideal values described above is referred to as an overhead in the plots.

[2]Note that using a different flag for each slave does not requires a lock acquiring operation

[3]On the Y-axis is plotted the number of cycles taken by barrier operations normalized to a single bus transaction cost (1 read + 1 write)

[4]The reader interested in code parallelization is referred to [6] and the references therein

[5]These routines essentially instantiate most of the synchronization structures in shared memory and link their lock field to some register of the hardware semaphores device

[6]So, for example, if eight processors are working on a vector of 32 elements, each core would process 4 data elements. The ideal execution time is that a single CPU would take to process a vector of 4 elements.

The fundamental parameters for our system are shown in table 1.

processor	ARM7, 200Mhz
data cache	4KByte, 4 way set associative latency 1 cycle
instruction cache	8KByte, direct mapped latency 1 cycle
private memory	latency 2 cycles
shared memory	latency 2 cycles
AMBA AHB	32 bit, 200Mhz, arbitration 2 cycles

Table 1: Architectural components details

5.1 Communication-Dominated Benchmark

The first set of experiments we wrote consists in square matrix filling (see below) and its purpose is that of investigating how concurrent accesses to shared regions limit the speedup of parallelization with the increasing number of cores. The number of rows, equal to the number of columns, has been parameterized with SIZE. Since each matrix element is written once and then never accessed again, we can expect that communication costs prevails on the benefits introduced by the parallelization.

```
for(i=SIZE*id/nprocs; i<SIZE*(id+1)/nprocs; i++)
  for(j=0; j<SIZE; j++)
    A[i][j] = 1.0;
```

The plot in figure 5 shows the results gathered for SIZE = 32. It shows the performance cost of the different contributions, intended as the overall number of cycles taken by each operation normalized by the number of cycles of an ideal bus transaction (1 read + 1 write). *Initialization* represents the number of normalized cost cycles needed for the library initialization routines to complete. This time grows with the number of cores because of the increasing bus contention. However, as initialization routines only occur at startup this cost is fixed and soon becomes negligible as overall execution time grows[7].

Synchronization time is split into two contributions: ideal synchronization time and relative overhead (the difference between measured and ideal time). The picture shows how the overhead grows with the number of cores. This is due to the additional bus traffic generated by the cores polling over shared synchronization structures. The amount of this overhead is very application-dependent, as the time the master will wait before all slaves enter the barrier is related to the synchronization pattern that the application shows at runtime.

Ideal parallel execution time (solid gray bars) follows the intuitive trend of almost halving with the doubling of the number of processors, but measurements show an overhead that severely limit the potential speedup.

The plot in figure 6 shows the results of the same benchmark but with a matrix of 1024x1024 elements. The cost relative to initialization completely disappears when compared to that relative to parallel execution, and so does synchronization. Parallel execution time overhead, on the other hand, has not disappeared. This is explained with the communication-dominated nature of the benchmark. A greater number of cores results in a greater number of concurrent shared memory access requests, that implies a lengthening of the the time needed to service all these serialized requests. Up to 4 cores the communication cost follows an almost-linear trend with a very small slope. With 8 cores or more this slope abruptly increases as bus contention produces very significant overheads. The scenario should change if we inserted more computation among bus accesses, and if we reduced these accesses to the minimum[8].

[7]Should this contribution still remain significant, a possible way to reduce bus contention would be that of serializing accesses. Limiting concurrent accesses to 2-4 cores per time would still take less time than the overhead.

[8]For instance declaring as cacheable the portion of the shared memory in which data resides.

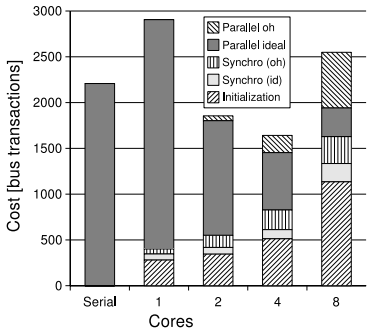

Figure 5: Performance results for communication dominated parallel execution (32x32 matrix filling).

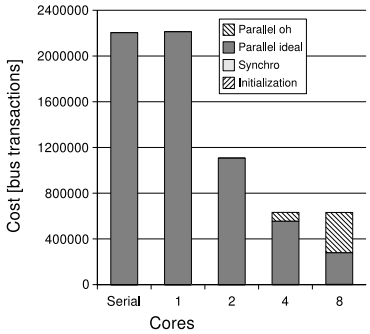

Figure 6: Performance results for communication dominated parallel execution (1024x1024 matrix filling).

5.2 Computation-Dominated Benchmark

The second set of experiments aims at investigating the convenience of parallelization in computation-dominated situations with very few accesses to shared resources. A vector is read in parts from shared memory, which is now declared cacheable, and a cycle of a variable number of iterations performs several sums on this data. Reducing the accesses to shared memory and performing a lot of computation on cached data we expect speedup to increase noticeably, as the cost for the sporadic bus accesses should go unnoticed if compared to that of elaboration. We made several different experiments, modifying both the number of elements of the shared vector and the number of iterations (i.e. the number of sums on data). We present here results for the configuration of 1024 elements vector and 1000 iterations.

```
for(i=0; i<ITERATIONS; i++)
  for(j=SIZE*id/nprocs; j<SIZE*(id+1)/nprocs; j++)
    tmp += A[j];

return tmp
```

Figure 7 shows the results for the execution of this benchmark. As expected, reducing accesses to shared memory the overhead measured in parallel computation completely disappears, thus confirming the hypothesis it was only due to data access-induced bus activity.

Figure 8 collects energy and performance results for 1000 iterations on a 1024 elements vector. Notice that speedup and additional energy follow curves with very different slopes. As speedup grows much more than consumed energy we can expect a big benefit in energy saving if voltage and frequency of cores is scaled[9].

[9]To give a quantitative estimation of the expected saving we can apply a simple, well known frequency scaling formula [14]

$$f = K\frac{(V-V_t)^2}{V}$$

where K is some constant of proportionality and $V_t = 0,7V$ is the

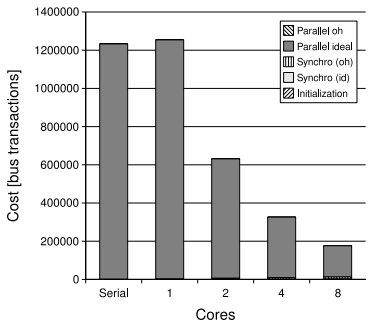

Figure 7: Performance results for computation dominated parallel execution (1024 elements vector x 1000 iterations).

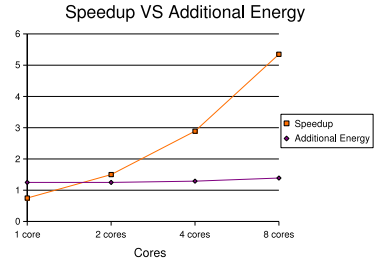

Figure 8: Comparison between energy and speedup results for the computation-dominated benchmark (1024 vector elements x 1000 iterations).

5.3 JPEG Decoding

In this section we present the results of a real multimedia benchmark: a parallelized version of the JPEG decoding algorithm. After the initialization, performed by every core, the master core starts computing the sequential part of the algorithm (Huffman DC and AC) while the slaves wait on a barrier. Then the computation is split between cores. Specifically each CPU applies on a slice of the reconstructed image a luminance dequantization and a reverse DCT filter. After each of these two parallel routines a barrier instruction is called, then the master can (optionally) compute a checksum on the decoded image. Data collected in figure 9 depict – as before – init, synchronization and parallel execution time with respective overheads. In addition the time taken by the master to execute the non-parallelized parts of the application is plotted as well. Due to the small number of barrier invocations synchronization time is negligible (as init time is). The behavior of the parallel portion of code follows the one we already discussed for the computation-dominated benchmark. During this section of the benchmark the cores access concurrently to shared data[10], and this limits the potential benefits of parallelization to the use of 4 cores. A larger number of CPUs would perform worse on this shared bus-based architecture. Another thing to point out is that the execution time of the sequential part increases drastically for more than 4 cores. This extra time is due to the slaves polling on shared variables while the master runs the sequential code. Since most modern MPSoCs are equipped with a tightly coupled memory (i.e. a scratchpad) a possible solution to eliminate this overhead would be that of instantiating in this local memory the synchronization structures. This will eliminate the extra overhead due to remote polling.

threshold voltage. If we substitute in this formula $f_1 = 200MHz$ (maximum speed) and $V_1 = 1,2V$ (unscaled CPU voltage) we can calculate $K = 960$. As from figure 8 we see speedup is 5,5x we can consider $f_2 = f_1/5,5$ as the speed at which we must run parallel code in order to complete in the same time taken by serial code. In this way from the formula is possible to determine the minimum voltage $V_2 = 0,89V$ allowed. We can conclude that

$$\frac{E_2}{E_1} \equiv \left(\frac{V_2}{V_1}\right)^2 = \left(\frac{0,89}{1,2}\right)^2 = 0,55$$

which confirms our hypothesis of a big potential energy saving.

[10] As already discussed allowing these data to be cached would reduce this overhead

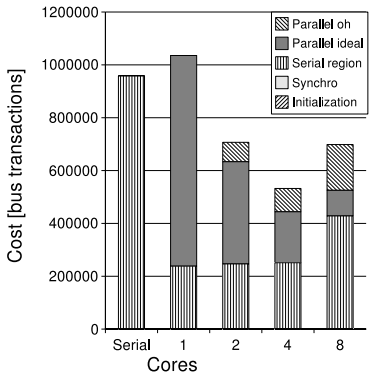

Figure 9: Performance results for parallel JPEG decoding

6. CONCLUSIONS

In this paper we presented a support library for the execution of compiler-generated parallel code on a MPSoC. Global synchronization is achieved by means of Master-Slave barriers, which implementation has been extensively evaluated using communication intensive and computation intensive benchmarks. Results are also given for a realistic application such as JPEG decoding. Our results indicate good scalability for up to 8 processors in computation-dominated situations, whereas in presence of high communication scalability is reduced to 4 processors. A detailed performance analysis shows that the main performance blocker is the bus contention. Hence our library never becomes the bottleneck in parallelization. This is a very promising results, and emphasizes the importance of low cost barriers with increased number of processors.

7. REFERENCES

[1] Jennifer M. Anderson and Saman P. Amarasinghe and Monica S. Lam, "Data and computation transformations for multiprocessors", In Proceedings of the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming, 1995

[2] Jennifer M. Anderson and Monica S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines", In PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, 1993

[3] Mary H. Hall and Saman P. Amarasinghe and Brian R. Murphy and Shih-Wei Liao and Monica S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler", In Supercomputing '95: Proceedings of the 1995 ACM IEEE conference on Supercomputing (CDROM), 1995

[4] Michael E. Wolf and Monica S. Lam, "A data locality optimizing algorithm", In Proceedings of the Conference on Programming Language Design and Implementation, 1991

[5] L. Xue and O. Ozturk and F. Li and and I. Kolcu, "Dynamic Partitioning of Processing and Memory Resources in Embedded MPSoC Architectures", In Design Automation and Test in Europe (DATE'06), Munich, Germany, 2006

[6] O. Ozturk and G. Chen and M. Kandemir, "A Constraint Network Based Solution to Code Parallelization", In Proc. Design Automation Conference (DAC)[Nominated for Best Paper Award], 2006

[7] Jong-eun Lee and Kiyoung Choi and Nikil D. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures", In LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language compiler and tool for embedded systems, 2003, pages 183–188

[8] C. Liu, A. Sivasubramaniam, M. Kandemir, M. J. Irwin, "Exploiting Barriers to Optimize Power Consumption of CMPs", In Proceedings of IPDPS, 2005.

[9] J. Sampson, R. Gonzàlez, J.F. Collard, N.P. Jouppi, M. Schlansker, "Fast Synchronization for Chip Multiprocessors", In ACM SIGARCH Computer Architecture News, 2005.

[10] J. Sampson, R. Gonzàlez, J.F. Collard, N.P. Jouppi, M. Schlansker, B. Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers", In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture MICRO 39, 2006.

[11] J. Li, J.F.Martìnez, M.C. Huang, "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors", In Proceedings of the 10th International Symposium on High Performance Computer Architecture HPCA '04 , 2004.

[12] S.Kumar, D.Jiang, R.Chandra, J.P.Singh, "Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance", In Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 1999

[13] J.M.Mellor-Crummey, M.L.Scott, "Algorithms for Scalable Synchronization on Shared Memory Multiprocessors", In ACM Trans. on Comp. Sys., 1991

[14] A.P.Chandrakasan, S.Sheng, R.W.Brodersen, "Low-Power CMOS digital design", IEEE Journal of Solid State Circuits, 1992

[15] MPARM Home Page, *www-micrel.deis.unibo.it/sitonew/mparm.html.*