A Scratch-Pad Memory Aware Dynamic Loop Scheduling Algorithm *

Ozcan Ozturk
Department of Computer Engineering
Bilkent University
Ankara, Turkey
ozturk@cs.bilkent.edu.tr

Mahmut Kandemir, Sri Hari Krishna Narayanan
Department of CSE
Pennsylvania State University
University Park, 16802, USA
{kandemir,snarayan}@cse.psu.edu

Abstract

Executing array based applications on a chip multiprocessor requires effective loop parallelization techniques. One of the critical issues that need to be tackled by an optimizing compiler in this context is loop scheduling, which distributes the iterations of a loop to be executed in parallel across the available processors. Most of the existing work in this area targets cache based execution platforms. In comparison, this paper proposes the first dynamic loop scheduler, to our knowledge, that targets scratch-pad memory (SPM) based chip multiprocessors, and presents an experimental evaluation of it. The main idea behind our approach is to identify the set of loop iterations that access the SPM and those that do not. This information is exploited at runtime to balance the loads of the processors involved in executing the loop nest at hand. Therefore, the proposed dynamic scheduler takes advantage of the SPM in performing the loop iteration-to-processor mapping. Our experimental evaluation with eight array/loop intensive applications reveals that the proposed scheduler is very effective in practice and brings between 13.7% and 41.7% performance savings over a static loop scheduling scheme, which is also tested in our experiments.

1 Introduction

Loop scheduling is an important component of any computing system that employs multiple processing units, e.g., chip multiprocessors. The basic functionality of this component is, for a loop-intensive application, to perform loop iteration-to-processor mapping so that a given loop nest can be executed in parallel to reduce execution latency. Depending on when this mapping is performed, a loop scheduling scheme can be described as static or dynamic. Static schedulers perform this mapping before the execution of the application starts, and therefore, cannot adapt well to dynamically changing execution conditions. Dynamic schedulers, on the other hand, decide this mapping at runtime, but, doing so requires a decision mechanism to be executed at runtime, which means an additional overhead in practice. In addition to these two main loop scheduling paradigms, prior work also explored the possibility of taking advantage of data reuse information to achieve better schedules than possible with pure static and dynamic schemes. Both static and dynamic schedulers have their advantages as well as drawbacks. Static schedulers are easy to implement and incur little overhead at runtime. However, they cannot capture runtime context well. In comparison, dynamic schedulers can capture runtime context (i.e., they can exploit the differences among the execution times of different iterations that belong to the same parallel loop) but are costly to implement and lead to extra runtime overheads.

A scratch-pad memory (SPM) is a small on-chip memory unit that is managed by software (e.g., through application program directly or via compiler-inserted code). They have low access latencies similar to conventional on-chip caches; however, they can be more energy-efficient than on-chip caches due to lack of dynamic tag-matching logic [5]. While there have been several attempts [10, 7] in the past to design data locality aware loop schedulers, these attempts target generally conventional data caches. In fact, the only SPM conscious loop scheduler we are aware of is [21], which is a static scheduler.

In this paper, we propose and experimentally evaluate an SPM-aware dynamic loop scheduling scheme for parallel computing systems. The main idea behind our approach is to take the access latencies of the SPM and off-chip memory into account explicitly so that the workloads of the parallel processors can be better balanced as compared to a conventional dynamic loop scheduler. To our knowledge, this new scheduler is the first dynamic scheduler that takes advantage of the SPM in performing the loop iteration-to-processor mapping at runtime. To study the behavior of this new scheduler, we implemented it using a publicly-available compilation framework and performed experiments with several array/loop intensive applications, targeting a chip multiprocessor system. In our experimental evaluation, we tested not only how this new scheduler compares to well-known static and dynamic schedulers, but also how much additional improvements it brings over three previously-proposed loop schedulers, which are either data locality oriented or SPM based. These experiments reveal that the proposed dynamic scheduler is very effective in practice and the performance gains it brings over a static loop scheduler vary between 13.7% and 41.7% (even when all the overheads incurred by our approach are included). In addition, we show that this approach can be used with the different variants of dynamic scheduling as well, such as tapering [13] and factoring [6]. Overall, our implementation and experimental evaluation not only show the practicality of building SPM-aware dynamic loop schedulers but also demonstrate that being SPM aware during loop iteration-to-processor mapping can be very important in scheduling array/loop intensive applications on parallel architectures.

The remainder of this paper is organized as follows. The next section gives a brief overview of existing static and dynamic loop scheduling techniques, including the most recent data locality aware schedulers as well as an SPM aware static scheduler. Section 3 presents the mathematical details of our proposed dynamic loop scheduler. Section 4 discusses our experimental results and explains our major findings. Section 5 concludes the paper.

2 Discussion of Related Work

Since there exist numerous studies on SPM management for both data and instruction accesses and this paper does not propose a new SPM management scheme, we do not discuss these SPM space management related efforts in this section, and refer the interested reader to [11, 21, 4, 5, 1, 15, 19, 3] and the references therein. Angiolini et al [2] present a fast SPM partitioning algorithm. Suhendra et al [16] combine SPM management and task scheduling targeting parallel architectures. As mentioned earlier, loop scheduling techniques can be divided into two main categories: static and dynamic. In static techniques, the iteration space of the loop to be executed in parallel across P processors is divided at compile time into P *sets,* (also called *chunks*) and each of these sets is assigned to a processor. Note that, while each set contains the same number of iterations, this does not mean that they will experience similar execution latencies at runtime. This is because two different iterations that belong to the same parallel loop can exhibit quite different data cache behaviors, execute the different branches of a conditional statement (i.e., different set of instructions), or contain sequential loops with different iteration counts (which means different workloads). All these factors (if they are severe) can eventually make a static scheduler very inefficient. In a dynamic loop scheduler, there is a master processor which assigns loop iterations to processors at runtime. The iteration space is divided into sets (chunks) of small sizes, and a processor that completes its current set asks for a new set from the master. In that way, a processor that finishes its load earlier than the others can ask for more work, thereby preventing it from waiting idle. Researchers have also studied several variants of this pure dynamic loop scheduling scheme. These variants include tapering [13], factoring [6], and trapezoidal loop scheduling [18], and they differ from the base dynamic scheduler in the sizes of the loads (in terms of loop iterations) assigned to processors at

*This work is supported in part by NSF grants CNS 0720645, CCF 0702519, and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRCs Focus Center Research Program.

0-7695-3117-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ISQED.2008.112

runtime (we will later give an example that demonstrates the differences between some of the variants of dynamic scheduling).

Apart from these two main scheduling schemes, previous research also investigated locality conscious and SPM aware loop scheduling schemes. In [10], Markatos et al discuss a locality aware loop scheduling scheme for conventional cache based systems. This approach takes advantage of a typical program behavior where the same parallel loop can be visited multiple times throughout the execution time of the program. This is really the case, for example, when a parallel loop is contained within an outer, sequential loop. The scheduling approach in [10] tries to ensure that a CPU is assigned the same set of loop iterations each time the same parallel loop is scheduled. In this way, the chances of catching the same data elements in the data cache are increased. More recently, [7] proposed a different locality aware scheduling scheme where the compiler selects the chunks of iterations to be assigned to a CPU carefully, based on the results of the loop-level data reuse analysis performed. In the SPM domain, recent work [21] proposes *a static* SPM scheduling algorithm oriented towards balancing the workloads across the CPUs taking into account the latency variation between an SPM access and an off-chip memory access. As compared to locality aware schedulers such as [10] and [7], our work is different as it targets an SPM based execution environment. In comparison to [21], on the other hand, our work, described in this paper, is a dynamic scheduler rather than a static one. In our experimental evaluation, in order to demonstrate the additional benefits brought by our dynamic loop scheduling approach over the state-of-the-art, we compare it to the loop schedulers published in [10], [7], and [21] as well.

3 Details of Our Approach

3.1 Architecture and Assumptions

While the loop scheduling approach explained below is applicable to a wide range of parallel systems, in this paper, our focus is on a chip multiprocessor with a shared on-chip SPM space. Note that what we mean here is a logical sharing of the available SPM space; this space can be physically distributed such that each processor is attached an SPM sub-space (a local SPM). Each SPM access is assumed to be much faster and takes much less energy to complete, as compared to an off-chip memory access. Whether this architecture has also instruction SPMs or not does not really concern us, as we are interested in only data SPM. We also assume that the SPM management decisions have already been made before our loop scheduler is invoked (i.e., before the master CPU gives a set of iterations to a requesting processor, it knows how many SPM accesses each iteration performs). Again, the exact algorithm used for deciding the contents of the SPM during the course of execution is orthogonal to the main focus of this paper, and in fact, our scheduler can work along with any SPM management algorithm. In particular, we want to emphasize that some SPM management schemes update the contents of the SPM dynamically at runtime. However, such updates are typically scheduled at compile time. Therefore, our dynamic scheduler can know the cost of every loop iteration (in terms of the number of SPM and off-chip memory accesses it makes) before it distributes, at runtime, chunks to CPUs. As a matter of fact, as will be discussed below, our proposed scheduling approach exploits this information about the relative costs of the different iterations that belong to the same parallel loop to better balance the workloads of the different CPUs, thereby (1) minimizing the number of times a CPU comes to the master processor asking for more work, and (2) reducing the overall execution time of the application by ensuring that each processor takes more or less the same number of cycles to finish.

Before discussing the details of our loop scheduling algorithm, we want to explain why SPMs make a good target for loop scheduling (as opposed to conventional hardware-managed caches). It is important to note that SPMs are managed by software, either by the application programmer manually, or by an optimizing compiler automatically. In either case, the contents of SPM at any given point in execution can be predicted since data transfers between the SPM and the off-chip memory are controlled by the software. As a result, one can estimate the cost of a given loop iteration as one can identify the number of SPM accesses and off-chip accesses that iteration will make at runtime. This estimation, while may not be extremely accurate, can be very useful in practice as it allows us to balance the workloads of different CPUs at compile time. As will be explained later in detail, this ability of estimating the costs of loop iterations and balancing workloads is an important component of the static portion of our loop scheduling algorithm. Its dynamic portion on the other hand adjusts the workloads at runtime to minimize overall application execution latency. Note that, since conventional data caches are managed by hardware such a cost estimation is not possible in general without being too conservative. Consequently, the use of static analysis for helping loop scheduling will be limited in practice when conventional cache memories are employed.

3.2 Compiler Analysis

The first component of the proposed approach is a static compiler phase, in which we assign a cost to each iteration of the loop to be executed in parallel by the CPUs in our chip multiprocessor. While this cost normally has both computation and memory access parts, we make our scheduling decisions based on only the memory access part because (1) it generally dominates computation cost and (2) computation cost does not change much from one loop iteration to another in our applications. Recall that we assume the SPM management decisions have already been made before our approach is applied. Hence, at any given point during execution we know the contents of the SPM. These contents are either compiler generated or specified by the application designer. Let us assume that a loop iteration i executes M_i memory access instructions, and an SPM access has a latency of L_{SPM} cycles, and a main memory access has a latency of L_{MEM} cycles, where $L_{SPM} \ll L_{MEM}$ in general (e.g., two orders of magnitude difference). The values of L_{MEM} and L_{SPM} are based on the architecture and hence are known at compile time. As far as the cost due to data memory access instructions is concerned, the cost of this loop iteration can be expressed as $M_{SPM_i}L_{SPM} + M_{MEM_i}L_{MEM}$, where $M_{SPM_i} + M_{MEM_i} = M_i$. It needs to be noted that M_{SPM_i} represents the number of accesses made by iteration i to the SPM, whereas M_{MEM_i} captures the number of accesses to the main off-chip memory by the same iteration. As the contents of the SPM are known at any point, the values of M_{SPM_i} and M_{MEM_i} are known as well. It is also important to note that this memory access cost can vary from one loop iteration to another, as a loop iteration can have different number of SPM and/or off-chip memory accesses than the other iterations that belong to the same parallel loop. In fact, the memory cost of a loop iteration can vary between M_iL_{SPM} and M_iL_{MEM}, depending on how many of its data requests are satisfied from the on-chip SPMs. If we use L^i_{TOT} to denote the memory cost of the ith iteration of the loop nest being analyzed, the total memory cost of executing this loop nest can be expressed as:

$$L_{TOT} = \sum_{i=1}^{N} L^i_{TOT} = \sum_{i=1}^{N} (M_{SPM_i}L_{SPM} + M_{MEM_i}L_{MEM}),$$

where N is the total number of iterations in the loop nest. Since the overall execution time of a parallel loop is determined by the latency incurred by the *latest* processor, it is very important to *balance* the execution latencies of the different processors. In other words, the execution latency of each processor should be as close to L_{TOT}/P as possible, where P is the number of CPUs in the chip multiprocessor architecture. Note that a formulation similar to the one above can also be written if there are latency differences between the different SPM components in the system (e.g., a local SPM can be faster to access than a remote SPM).

As explained earlier, a classical dynamic loop scheduler addresses this load balancing problem by distributing the loop iterations to CPUs at runtime. In fact, the only difference among the different dynamic scheduling schemes proposed in literature is in the number of loop iterations given to a CPU when it requests more work during the parallel execution of a loop. In the baseline dynamic scheduler known as the self-scheduler, the iteration space is divided into L chunks of the same size (in terms of the number of iterations), where $L \gg P$ (i.e., the number of groups is larger than number of CPUs). Each time a processor finishes its work and asks for more, it is given one of these

L chunks, which have not been processed so far. In tapering, a variant of the baseline dynamic scheduling, the size of each chunk of iterations given to a processor (when it requests more work) is always $1/P$ of the remaining loop iterations to be executed. Under this scheduling policy (which is also referred to as the guided self-scheduling in some research papers [13]), it is guaranteed, under certain circumstances, that all the CPUs will finish within one iteration of another. Another variant of the baseline dynamic scheduling is called factoring, which is similar to tapering but, instead of computing the size of one chunk of iterations at each scheduling step, it computes the size of a batch of chunks. However, since all these schemes work with the number of iterations rather than predicted memory access latencies, the resulting workloads assigned to processors may not be well balanced, which means (1) processors frequently ask the master CPU to give them more work and (2) one or more of the processors can finish their last work much earlier than the others, increasing the overall execution latency (an example will be given in Section 3.3 to make this issue clear).

Our approach, instead, tries to balance the workloads across the CPUs as much as possible, which helps to address the two problems mentioned above. Our goal is to eliminate the two problems, mentioned above, associated with the conventional dynamic schedulers. Suppose that, at some point during execution, we want to schedule a set S of iterations across P processors. We use $L_{TOT}(S)$ to denote the total access latency of these loop iterations. In other words[1],

$$L_{TOT}(S) = \sum_{i=1}^{|S|} L^i_{TOT}.$$

Our goal is to divide these $|S|$ iterations into P sets (T_1, T_2, $\cdots$, T_P) such that

$$\sum_{i \in T_1} L^i_{TOT} \approx \sum_{i \in T_2} L^i_{TOT} \approx \cdots \approx \sum_{i \in T_P} L^i_{TOT} \approx L_{TOT}(S)/P,$$

where $T_1 \cup T_2 \cup \cdots \cup T_P = S$ and $T_1 \cap T_2 \cap \cdots \cap T_P = \emptyset$. We now describe how our SPM aware scheduling approach achieves such a balanced distribution, and in Section 3.3, we show an example application of our approach.

We use Q_i to denote the iterations in S which have the same latency (i.e., the iterations in a Q_i need the same number of SPM and off-chip memory accesses). In other words, the iterations in different Q_i sets have different latencies. Since each Q_i is a subset of S, we have:

$$\bigcup_{i=1}^{m} Q_i = S.$$

Note that the Q_i sets are identified at *compile time,* and this forms the static portion of our approach. Our scheduling algorithm then builds, *at runtime,* each T_i set by assigning $|Q_j|/P$ iterations to T_i from each and every Q_j, where $1 \leq j \leq m$ and m represents the number of different latencies. In other words, our T_i set construction is based on satisfying the following constraint:

$$|T_i| \approx \sum_{j=1}^{m} |Q_j|/P.$$

This ensures that the iterations having the same latency are evenly distributed to all the available processors, that is P sets ($T_1, T_2, \cdots, T_P$). Since we repeat this step until all Q_i sets are processed, the resulting loop distribution will satisfy our goal explained above.

At this point, we want to explain why our scheduling approach is a dynamic one, as opposed to a static one. It is true that, using a static compiler analysis, we first group the iterations of the loop to be scheduled into latency classes (the Q_i sets, as explained above). However, the actual assignment of these iterations to CPUs is performed at runtime based on the explanation given above (Figure 1 gives the pseudo code that makes this iteration-to-CPU assignment). This is because, although the compiler can estimate the cost of a loop iteration, this estimation may not be very accurate, due to several reasons (e.g., increased memory access latency due to contentions). Also, some of the iterations may simply be dropped as a result of some conditionally-executed constructs (e.g., if statements). If we perform the iteration assignment at compile time, we could not take advantage of this runtime information. In our implementation, whenever there is a change detected (at runtime) in iteration latencies, this information is taken into account and the Q_i sets are updated accordingly. Therefore, our approach performs iteration assignment at runtime and, thus, it is dynamic.

[1] Given a set S, we use $|S|$ to denote the cardinality of S.

```
P: the number of processors available for scheduling.
T_k: the set of iterations being assigned to processor k.
Q_j: the set of iterations exhibit same latency.
S: the set of iterations to be scheduled, i.e., S = ⋃_{j=1}^{m} Q_j.

For each T_k, where 1 ≤ k ≤ P, initialize T_k to ∅.

While (S != ∅)
  For each k, where 1 ≤ k ≤ P
    For each j, where 1 ≤ j ≤ m
      While (Q_j != ∅)
        s_j = the next |Q_j|/P iterations from Q_j.
        Q_j = Q_j – s_j. /* update Q_j */
        S = S – s_j. /* update S */
        T_k = T_k ∪ s_j.
      End while
    End for
  End for
End while
```

Figure 1. Our loop scheduling algorithm in pseudo code.

We also want to clarify the issue of how the dependencies are handled. Our current implementation does not parallelize a loop that carries any data dependence, and among the loops (in a nest) that can run parallel, we select the outermost one to minimize the potential synchronization costs (or try to interchange loops such that the parallel loop is placed in the outermost position). However, if desired, our approach can be modified to work with the cases where loops that are not entirely dependence-free are parallelized. In this case, our loop iteration costs should be updated with the necessary interprocessor synchronization costs. An analysis of our benchmark codes showed however that in nearly 82% of the loop nests, our compiler was able to identify at least a loop that is dependence-free. In addition, in about 95% of such cases, we were able to bring that (parallel) loop to the outermost position, thereby minimizing the synchronization overheads among the parallel CPUs.

3.3 Example

To see the differences between the base dynamic scheduler and its two variants (tapering and factoring) as well as the SPM aware versions of these schedulers, let us consider a scenario where 60 iterations (for illustrative purposes) of a parallel loop are to be scheduled across 4 processors. Table 1(a) shows the iteration assignments with three conventional dynamic schedulers. In this example, the chunk size used for Dynamic is 6 and factoring value, f, used for Factoring is 2. It is important to note that none of these three schedulers are SPM aware. Consequently, the real execution times of any two chunks of the same size (in terms of iterations) can be dramatically different from each other. Assuming the iteration latencies given in Table 1(b), the execution times of the four processors are shown as in Table 1(c). For illustrative purposes, we assume that there are 2 cycles for SPM access and 90 cycles for off-chip memory access. For instance, if there

are 2 SPM accesses and 3 off-chip memory accesses in a particular loop iteration, the latency of that iteration is estimated to be 274 (2*2+3*90) cycles. In Table 1(c), for each processor, we give the execution latency for each scheduling step. For example, in Dynamic scheduling, the cell that contains 8+98+10 (116) means that the total latency of the third CPU is 116 at that particular scheduling step, and this value (116) results from the contributions of three different latency groups of iterations. For each scheduling scheme, the largest latency value is given in bold. For example, in Dynamic scheduling, the overall execution latency is 752. Since the overall execution latency of the parallel loop is dictated by the slowest processor, we can estimate the execution latencies of this example under the Dynamic, Tapering, and Factoring scheduling schemes to be 752 cycles, 758 cycles, and 948 cycles, respectively, without taking into account the bookkeeping performed by the master CPU and other CPU related activities. Now, let us look at the execution latencies under these three dynamic scheduler variants when they are made SPM aware using our proposed approach. The processor latencies (cycles) in this case are given in Table 1(d). We see that the execution latencies under the SPM aware versions of Dynamic, Tapering, and Factoring scheduling schemes to be 596 cycles, 598 cycles, and 590 cycles, respectively. When compared with the results obtained by using the corresponding SPM oblivious versions, we observe significant performance improvements (20.7% in Dynamic; 21.1% in Tapering; and 37.8% in Factoring to be exact).

4 Experimental Evaluation

4.1 Setup

We used a SIMICS [14] based simulation platform to make our experiments. We first enhanced SIMICS with accurate timing models and then simulated a chip multiprocessor, where each processor has a local SPM space. Each processor can access local SPM spaces of other processors as well. A compiler pass, applied prior to our scheduling, decides the contents of the SPMs at any given point in the program code, using the approach described in [1]. In this SPM management approach, first, a data reuse analysis is performed, and then, the data transfers between the SPM and the off-chip memory are scheduled based on the results of this data reuse analysis. The goal of this scheme is to displace a block of data from the SPM once its temporal reuse is over. While we used this particular SPM management scheme in our experiments, nothing prevents us from using our dynamic scheduling scheme with other SPM management strategies as well. This SPM management part and the static compiler part of our approach have been implemented using the SUIF infrastructure [8]. Note that our simulator simulates, in detail, the activities of both the master processor and the processors that execute loop iterations in parallel and also records *all the overheads involved*, due to dynamic scheduling.

For each benchmark code in our experimental suite, we performed experiments with different versions. Specifically, in this section, Dynamic, Tapering and Factoring refer, respectively, to the baseline dynamic loop scheduler, its tapering version, and its factoring version, respectively. The specific implementations used for Dynamic, Tapering and Factoring are, respectively, from [17], [13], and [6]. We also report results when these three major dynamic scheduling schemes are made SPM aware using the approach proposed in this paper. LA-I and LA-II represent the loop scheduling schemes described in [10] and [7], respectively. The detailed discussions of these schemes are given earlier in Section 2. In addition, we also compare our approach to the only SPM aware loop scheduling scheme we are aware of [21], whose details are also discussed in Section 2. As mentioned earlier, the scheme in [21] is a purely static loop scheduling scheme.

Table 2 gives the default simulation parameters used in most of our experiments, and Table 3 lists the benchmark code used for evaluation of the different scheduling schemes. The second column presents a brief description of the benchmark and the next column gives the amount of data processed by it. The last column gives the execution cycles under a static loop scheduling scheme. In this scheme, the iteration space of a parallel loop is divided into P even (or nearly even) sets, and each processor is assigned a set; this partitioning takes place at compile time.

Dynamic	6	6	6	6	6	6	6	6	6	6
Tapering	15	12	9	6	5	4	3	2	1	1	1	1
Factoring	8	8	8	8	4	4	4	4	2	2	2	2
	1	1	1	1								

(a) Iteration assignment under three dynamic loop scheduler.

Iteration #	2	4	10	14	18	21	27	28	30	34	39	47	48	50	53	58
Latency	10	98	274	98	10	98	10	186	274	186	274	186	274	10	98	186

(b) Latency of some of the iterations; all other iterations are assumed to have 2 cycles latency.

Dynamic	P1	8+10+98 (116)	10+186 (312)	8+10+98 (428)	
		10+274 (284)	8+186+274 ( )		
	P3	8+98+10 (116)	10+274 (400)	10+186 (596)	
	P4	10+98 (108)	6+10+186+274 (584)		
Tapering	P1	22+10+98+274+98 (502)	4+98 (604)	2 (606)	
	P2	18+10+98+10 (136)	8+186 (330)	2 (332)	
	P3	12+186+274+186 (658)	4 (662)	2 (664)	
		10+274 (284)	4+274+10 (572)	186 ( )	
Factoring	P1	12+10+98 (120)	6+186 (312)	2+10 (324)	2 (326)
	P2	12+274+98 (384)	8 (392)	4 (396)	186 (582)
	P3	12+10+98 (120)	6+274 (400)	2+98 (500)	2 (502)
		10+10+186+274 (480)	4+186+274 (942)	4 (946)	2 ( )

(c) Execution times under the conventional approaches.

SPM Aware Dynamic	P1	10+98+186+274+4 (572)	12 (584)	12 (596)	
		10+98+186+274+4 (572)	12 (584)	12 ( )	
	P3	10+98+186+274+4 (572)	12 (584)		
	P4	10+98+186+274+4 (572)	12 (584)		
SPM Aware Tapering	P1	10+98+186+274+22 (590)	4 (594)	2 (596)	
	P2	10+98+186+274+16 (584)	6 (594)	2 (596)	
		10+98+186+274+10 (578)	8 (596)	2 ( )	
	P4	10+98+186+274+4 (572)	10 (582)	2 (584)	
SPM Aware Factoring	P1	10+98+186+274+8 (576)	8 (584)	4 (588)	2 (590)
	P2	10+98+186+274+8 (576)	8 (584)	4 (588)	2 (590)
	P3	10+98+186+274+8 (576)	8 (584)	4 (588)	2 (590)
		10+98+186+274+8 (576)	8 (584)	4 (588)	2 ( )

(d) Execution times under our SPM aware approaches.

Table 1. Example.

4.2 Results

Before presenting the performance improvements provided by our proposed scheduling approach, let us see how much benefit an SPM oblivious dynamic loop scheduling and its variants (tapering and factoring) can bring. We give in Figure 2 the execution latencies of our benchmarks under these three dynamic schemes on an 8 CPU chip multiprocessor, *normalized* with respect to the latency of the static loop scheduling. That is, for each benchmark, the execution latency of the static loop scheduling scheme is set to 100, and the rest of the results are normalized with this value. An interesting trend we observe from these results is that none of the variants of the dynamic scheduling provides any performance improvement over the static scheduling scheme. The main reason for this is that these schedulers treat all iterations of a given parallel loop the same, whereas in reality each iteration can have significantly different execution latency than the others, depending on how many SPM and off-chip memory accesses it performs as well as the relative latencies (in terms of clock cycles) of the SPM and off-chip accesses (as illustrated by the example discussed in Section 3.3). In addition, dynamic scheduling of loop iteration at runtime itself brings a certain degree of overhead, as the iteration distribution is carried out at runtime. More imbalance the workload distributions exhibit, more frequently the master CPU is required to perform workload distribution. These factors together

Simulation Parameter	Default Value
Number of Processors	8
IPC	2
SPM (Per Processor)	8KB
SPM Access Latency	1 Cycle (local); 3 cycles (remote)
On-Chip Memory Access Latency	100 Cycles

Table 2. Our main simulation parameters and their default values.

Benchmark Name	Brief Description	Dataset Size	Execution Latency
Med-Im04	Medical Image Reconstruction	0.79MB	1.36sec
Radar	Radar Imaging	3.83MB	3.54sec
Shape	Pattern Recognition and Shape Analysis	3.28MB	2.97sec
Track	Visual Tracking Control	1.09MB	1.88sec
Sparrow	Image Extraction	2.84MB	3.15sec
Ecode	Image Encoding/Decoding	5.16MB	4.24sec
Rod	Video Frame Reordering	4.91MB	3.72sec
Balls	Game Application (Bouncing Balls)	2.27MB	2.96sec

Table 3. Our applications.

make classical dynamic scheduling and its variants unattractive for an SPM based execution platform. Consequently, we observe that Dynamic, Tapering and Factoring degrade the original performance of the static scheduler by 3.2%, 12.6% and 12.9%, respectively, on average. Figure 2 also shows the optimal execution latencies of our benchmarks. These values, which are captured by the last bar of each benchmark in the graph, are calculated assuming that each processor experiences the same (ideal) latency, L_{TOT}/P. Note that this may or may not be realizable, depending on the SPM and off-chip memory latency values as well as the number of CPUs in the system and the number of loop iterations to be scheduled. We see that the optimal scheme improves over the static scheme by 33.8% on average. Overall, these results given in Figure 2 clearly indicate that there is a large potential for improving the behavior of the static scheduling scheme (as made evident by the results of the optimal scheme) but the traditional dynamic loop scheduling schemes fail to achieve this potential.

We now present the results when our scheduling approach is applied. As noted earlier, our approach can work with any dynamic loop scheduling scheme. For a benchmark, the first three bars in Figure 3 give the results of our approach when it is applied to Dynamic, Tapering and Factoring; these new versions are named as Dynamic+, Tapering+, and Factoring+ in the graph. The fourth bar of this graph on the other hand reproduces the results of the optimal scheme for ease of comparison. Again, the results are normalized against the static scheduling scheme. One can clearly see, when comparing these results with those in Figure 2, that making these dynamic scheduling schemes SPM aware has a significant impact on our results. In fact, the new schemes Dynamic+, Tapering+, and Factoring+ improve over the performance of the static scheduler by 26.4%, 27.4%, and 27.4%, respectively, on average. Recalling that the average improvement brought by the optimal scheme was 33.8%, we can conclude that our approach performs really well in practice. However, for completeness, we need to compare these enhanced dynamic scheduling schemes to the previously-proposed locality aware and SPM conscious scheduling schemes as well. In Figure 3, the last three bars for each benchmark give the normalized execution latencies of the schemes LA-I, LA-II and SPM aware (static scheduling). The average performance improvements brought by these schemes are 4.5%, 10.8%, and 22.4% in that order. As mentioned earlier, the LA-I scheme relies on the observation that a given parallel loop can be visited multiple times during the execution of the application program. Consequently, if each processor gets the same set of iterations in successive invocations of the parallel loop, we might be able to improve data locality. In the context of an SPM based architecture, this means reusing the data in the local SPM rather than in other SPMs in the system. As can be seen from our results given in Figure 3, this approach

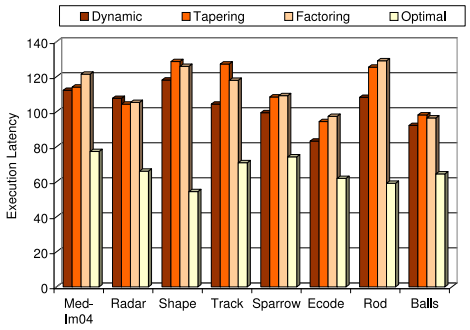

Figure 2. Execution latencies of the conventional dynamic scheduling schemes normalized with respect to that of the static scheduling scheme. The last bar, for each benchmark, represents the optimal results.

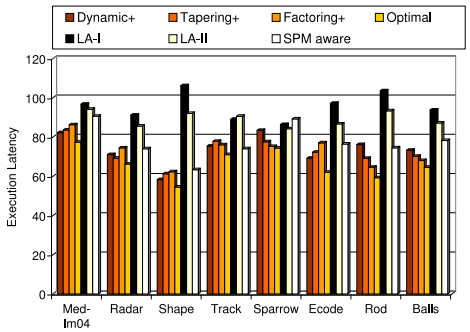

Figure 3. Execution latencies of the different scheduling schemes normalized with respect to that of the static scheduling scheme. The last bar, for each benchmark, represents the optimal results.

generates good results for some of our benchmarks such as Track and Sparrow, where the parallel loops are indeed revisited multiple times. However, in some other benchmarks such as Shape and Rod, where this is not the case, this approach does actually degrade performance, since it is in essence a static scheme and cannot capture the dynamic workload variances at runtime. In comparison, LA-II performs better than LA-I, since it is more general (i.e., not just restricted to the loops that are visited multiple times during execution), except for one benchmark (Track), and it captures dynamic variances as well. The SPM aware static scheduling scheme (denoted as SPM aware in the graph) on the other hand is generally more successful than both LA-I and LA-II, except for the benchmark Sparrow. This is because this scheme takes SPM latencies into account; however, it does not perform as well as our approach in general, mainly due to the fact that it is a pure static approach and the contents of the SPMs (as well as the estimated latencies of loop iterations) can change at runtime based on the changes in data reuse characteristics.

Figure 4 presents the results of our scalability analysis with the different loop scheduling schemes implemented in this work. We varied the number of processors (on the x axis) from 4 to 18, and measured the percentage performance improvements over the static scheduling scheme. A negative value in this graph corresponds to a performance degradation. Maybe the most important observation to be made about these results is the very poor scalability of the conventional dynamic schemes, and the significant improvements obtained when these schemes are operated under our SPM aware approach. The difference the SPM awareness makes gets magnified as we increase the number of CPUs in the chip multiprocessor. Thus, these results clearly show the importance of taking into account the SPM when scheduling loop iterations at runtime.

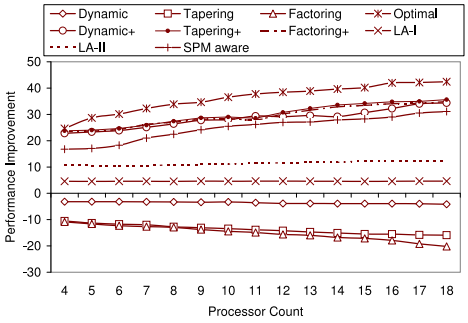

Figure 4. Scalability analysis.

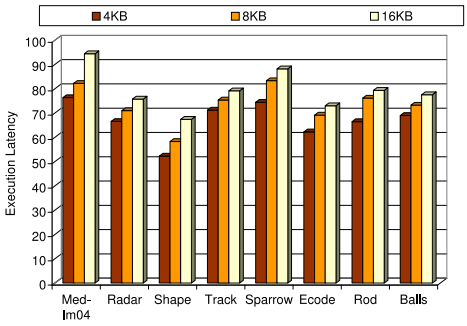

Figure 5. Results with the different SPM capacities.

Figure 5 gives the execution latencies (again, normalized with respect to the static loop scheduling scheme) with the different SPM capacities (per CPU). The default SPM capacity used in the experimental evaluation so far was 8KB per CPU (see Table 2) and the results with this are also reproduced in Figure 5 for comparison purposes. Since the trends with the Dynamic+, Tapering+, and Factoring+ schemes are all similar to each other, this graph shows only the results with Dynamic+. As can be observed, while our approach performs best with the smallest SPM capacity (since, as the SPM capacity gets smaller, it is becoming even more important to employ SPM aware scheduling), even with the 16KB (per CPU) SPM, our approach achieves significant savings. The average performance improvements with 4KB, 8KB and 16KB SPMs are 32.6%, 26.4% and 20.6%, respectively.

Our final analysis studies the impact of our loop cost estimation approach. Recall from Section 3.2 that our default cost estimation analysis considered only the SPM and off-chip memory accesses to estimate the overall cost of a loop iteration. We also implemented a more sophisticated estimator which accounts for CPU cycles as well. To do this, the compiler counts the number of operations of different types (such as addition, comparison, etc) and, taking into account data dependencies (to compute memory waiting latencies) and issue widths, estimates the cycles that will be consumed by the CPU. The details of this estimation are beyond the scope of this paper, but our overall estimation algorithm is similar in spirit to the approach explained in [20]. The results with this more powerful estimator are presented in Figure 6 (the results with the default estimator are reproduced in this plot for ease of comparison). We see that the simple cost estimation performs reasonably well (i.e., the results with two estimators are very close to each other). Given also the fact that this more accurate estimator increased compilation times by 4 to 5 times over the simple estimator, we believe that the latter is a better choice.

5 Conclusions

Execution time of a loop based application code on a chip multiprocessor is strongly influenced by the effectiveness of the loop scheduler, whose main task is to assign loop iterations to CPUs for execution. In an SPM based execution platform where two iterations of the same parallel loop can have dramatically different execution latencies due to the disparity between data access latencies of the SPM and the off-chip memory, not taking into account the memory access latencies can lead to poor scheduling decisions. In this paper, we present and experimentally evaluate an SPM aware dynamic loop scheduling scheme in the context of chip multiprocessors. To test the impact of this new scheduler, we implemented it and performed experiments with several array/loop-intensive applications. In our experimental evaluation, we tested not only how this new scheduler compares to well-known static and dynamic schedulers, but also how much additional improvements it brings over three previously proposed loop schedulers, which are either data locality oriented or SPM based. These experiments reveal that the proposed scheduler is very effective in practice and the additional performance gains it brings over the second best scheduler tested during our experiments are significant. To sum up, our implementation and evaluation not only shows the practicality of building SPM-aware dynamic loop schedulers but also demonstrates that being SPM aware during loop iteration-to-processor mapping can be very important in scheduling array/loop intensive applications.

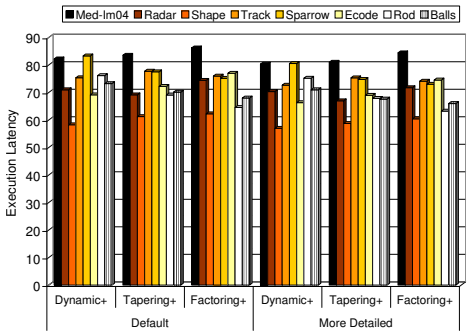

Figure 6. Comparison of two cost estimation approaches.

References

[1] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch-pad memory space in embedded multiprocessor systems, In *Proc. of DAC*, June 2002.

[2] F. Angiolini , L. Benini , and A. Caprara. Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning. In *Proc. of CASES,* San Jose, CA, 2003.

[3] O. Avissar , R. Barua , and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Computing Systems,* 1(1):6-26, November 2002.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proc. of CODES*, May 2002.

[5] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing Energy Efficiency of Embedded systems by Application-Specific Memory Hierarchy Generation. *IEEE Design & Test of Computers,* pages 74–85, April-June, 2000.

[6] S. F. Hummel, E. Schonberg, and E. F. Lawrence. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, August 1992, 35(8):90-101.

[7] M. Kandemir. LODS: Locality-Oriented Dynamic Scheduling for On-Chip Multiprocessors. In *Proc. of DAC*, June 2004.

[8] S.-W Liao, A. Diwan, R. P. Bosch, Jr. and A. Ghuloum, and M. S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In *Proc. of PPoPP,* May, 1999.

[9] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proc. of PLDI*, June 1992.

[10] E. P. Markatos, T. J. LeBlanc. Using Processor Affinity in Loop Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design and Test Conference*, March 1997.

[12] C. D. Polychronopoulous. *Parallel Programming and Compilers.* Norwell, Massachusetts, Kluwer Academic Publishers.

[13] C. D. Polychronopoulous, D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 1987.

[14] *SIMICS Simulator.* http://www.virtutech.com/

[15] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of DATE,* 2002.

[16] V. Suhendra, C. Raghavan, and T. Mitra. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architecture. In *Proc. of CASES,* 2006.

[17] T. H. Tzen and L. M. Ni. Dynamic Loop Scheduling on Shared-Memory Multiprocessors. In *Proc. of ICPP,* Vol. II, 1991, pp. 247-250.

[18] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling Scheme for Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[19] L. Wang, W. Tembe, and S. Pande. A Framework for Loop Distribution on Limited On-Chip Memory Processors. In *Proc. of CC,* March 2000.

[20] M. Wolf, D. Maydan, and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proc. of MICRO,* 1996.

[21] L. Xue, M. Kandemir, G. Chen, and T. Yemliha. SPM-Conscious Loop Scheduling for Embedded Chip Multiprocessors. In *Proc. of ICPADS*, July 2006.