Memory Bank Aware Dynamic Loop Scheduling*

Mahmut Kandemir† Taylan Yemliha‡ Seung Woo Son† Ozcan Ozturk†
†Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{kandemir,sson,ozturk}@cse.psu.edu
‡College of Engineering and Computer Science
Syracuse University
Syracuse, NY 13244, USA
tyemliha@syr.edu

Abstract

In a parallel system with multiple CPUs, one of the key problems is to assign loop iterations to processors. This problem, known as the loop scheduling problem, has been studied in the past, and several schemes, both static and dynamic, have been proposed. One of the attractive features of dynamic schemes, as compared to their static counterparts, is their ability of exploiting the latency variations across the execution times of the different loop iterations. In all the dynamic loop scheduling techniques proposed in literature so far, performance has been the primary metric of interest. In a battery-operated embedded execution environment, however, power consumption is another metric to consider during iteration-to-processor assignment. In particular, in a banked memory system, this assignment can have an important impact on memory power consumption, which can be a significant portion of the overall energy consumption, especially for data-intensive embedded applications such as those from the domain of image data processing. This paper presents a bank aware dynamic loop scheduling scheme for array-intensive embedded media applications. The goal behind this new scheduling scheme is to minimize the number of memory banks that need to be used for executing the current working set (group of loop iterations) when all processors are considered together. That is, during the loop iteration-to-processor assignment, our approach considers the bank access patterns of loop iterations and carefully selects the set of iterations to assign to an idle processor so that, if possible, the number of memory banks that are used at the current state is not increased. Our experimental results show that the proposed scheduling scheme leads to much better energy results when compared to prior loop scheduling techniques and it is also competitive with the scheduler that generates the best performance. To our knowledge, this is the first dynamic loop scheduling scheme that is memory bank aware.

1. Introduction

Loop scheduling is the process of assigning iterations of a parallel loop into processors and has been studied in the past for various types of parallel architectures. Previously published loop scheduling techniques can be roughly divided into static and dynamic techniques. The static scheduling techniques try to perform the loop iteration-to-processor assignment at compile time before the applications starts executing, whereas the dynamic techniques postpone this assignment to runtime, in an attempt to take dynamic variations across the execution times of the different loop iterations into account and reach well balanced loads across the different processors by exploiting these variations. These dynamic variations, as will be discussed shortly, can occur due to different reasons such as conditional flow of control and cache behavior.

Dynamic loop scheduling has been studied in the past from the perspectives of load balance and data locality, both of which are oriented towards improving performance. As compared to static loop scheduling techniques, its main advantage is the ability of capturing the variations across the workloads of different CPUs and exploiting this information when performing iteration assignment at runtime. However, in a battery-operated embedded execution environment, power consumption is another metric to consider during this iteration assignment. In particular, in a banked memory system, the loop iteration-to-processor mapping can have substantial impact on memory system power consumption (since it determines the set of banks that will be exercised at any given period of time), which can be a significant portion of the overall energy consumption for the data-intensive embedded applications.

The main contribution of this paper is a *bank aware dynamic loop scheduling scheme* for array-intensive embedded media applications. The goal behind this new loop scheduling scheme is to minimize the number of memory banks that need to be used for executing the current working set (group of loop iterations) when all processors are considered together. The unused memory banks can then be held in a low power state, for longer periods of time, to conserve energy. The proposed scheduling approach represents both the current active/idle status of banks and the set of banks that may be accessed by a given loop iteration using bitmaps and uses these bitmaps at runtime in performing the iteration assignment. The goal is to minimize the number of banks that are active at any given period of time.

We implemented this memory bank-aware loop scheduling scheme and performed experiments with several embedded application codes. In our experimental evaluation, we compare it, in terms of both energy consumption and performance, to a number of previously proposed loop scheduling strategies, including both static and dynamic techniques. Our experimental results show that the proposed scheduling scheme not only reduces the energy consumption significantly, but it also leads to much better energy savings when compared to these prior techniques and it is competitive with the loop scheduler that generates the best performance. To our knowledge, this is the first dynamic loop scheduling scheme that is memory bank aware.

The remainder of this paper is structured as follows. The next section gives background on loop scheduling and banked memories. It also discusses the relevant prior work on loop scheduling. Section 3 presents the details of our proposed bank-aware loop scheduling scheme. This scheme is evaluated along with several previously proposed scheduling techniques in Section 4, and the results are discussed from both energy and performance angles. Section 5 concludes the paper and outlines the possible future research directions.

2. Background on Loop Scheduling and Banked Memories

As mentioned earlier, the prior work on loop scheduling considered both static and dynamic techniques. The static techniques perform iteration assignment to CPUs at compile time, and therefore, are easy to implement, as compared to their dynamic counterparts, which require some work to be performed at runtime, thereby contributing (as overhead) to the execution time. The basic static technique [14, 19] divides the iteration space of the parallel loop (i.e., the set of all iterations in the loop) to be scheduled into P equal (or almost equal) subsets where P is the number of processors at hand, and each processor is assigned a subset. As men-

*This work is supported in part by NSF grant #0093082 and a grant from GSRC.

978-3-9810801-2-4/DATE07 © 2007 EDAA

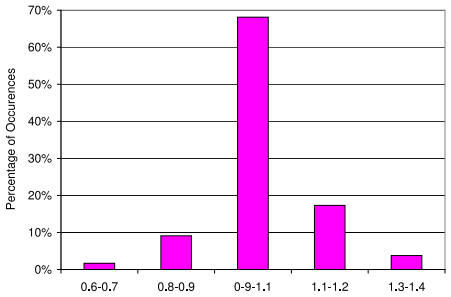

Figure 1. Distribution of the execution latencies of iterations for a typical parallel loop from one of our applications.

tioned in [9], static scheduling can also assign iterations in a locality aware fashion by ensuring that the set of iterations assigned to a processor exhibit data reuse among themselves. However, such a purely static assignment of iterations can lead to load imbalances, which can be due to the different reasons:

- Variations due to conditional control flow. For example, the different branches of an IF or SWITCH statement can be taken by the different loop iterations, and consequently, there can be large variations across the execution times of the iterations that belong to the same parallel loop.
- Variances due to loop index dependent bounds. When the lower bound and/or upper bound of an inner (sequential) loop depends on the index of the parallel outer loop, the different iterations of this outer loop can experience different execution latencies.
- Variances due to data locality (cache behavior). The different loop iterations can produce different cache hit/miss counts, which can in turn lead to significant variations among their execution times.

It needs to be noted that most of these variations are not possible to capture at compile time, and thus, a purely static scheduling technique can be very inefficient when these variations are really significant. As a result, due to these factors, partitioning loop iteration space across available processors evenly such that each processor receives more or less the same number of iterations can lead to large variances across the execution times of the CPUs. In addition, since the execution time of a parallel loop nest is typically determined by the execution time of the processor that finishes its group of iterations last, an unbalanced partitioning (in terms of execution cycles) can be detrimental to performance. To give an idea about the magnitude of this variance, Figure 1 shows the distribution of the execution latencies of iterations for one of the loops in baleen, one of our embedded applications (we will present the details of our embedded applications and the simulation platform used later). The y-axis in this figure captures the percentage of occurrences for each latency group. The value 1 on the x-axis represents the most frequently occurring latency across all iterations, and all other latencies are normalized with respect to that value. So, each bar gives the percentage of occurrences for a latency interval. It is easy to see that the largest variance across the executing latencies of the different loop iterations is around 40%, indicating the severity of the problem. We need to mention that this particular loop nest was not an extreme case; rather, it was exhibiting a typical behavior. In fact, we observed during our experimental evaluation that the largest variation across two different iterations of the same parallel loop varied between 1% and 82%, averaging on 37%.

In dynamic loop scheduling, a master CPU assigns works to slave (worker) processors at runtime. One of the earliest dynamic scheduling schemes is known as self scheduling [14], and addresses load imbalance by assigning a small load to slave processors initially, and allowing the slaves to ask for more work once they are done with their current assignments. Each work assignment takes place within a critical section, a piece of code for which entrance is granted to one CPU at a time. A variant of self scheduling is tapering, also known as guided self scheduling [15], where the loads assigned to processors are reduced gradually as the execution progresses, in an attempt to prevent potential imbalances towards the end of the iteration space. The other variants of dynamic scheduling include factoring [6] and trapezoid self scheduling [18]. Our initial experiments with these different performance overhead of dynamic schedulers found that the performance and energy behaviors of tapering, factoring, and trapezoid self scheduling were very similar to each other (within 1% of one another). Therefore, as far as dynamic schedulers are concerned, in this paper, we present results with self scheduling and tapering only.

More recently, there have been several proposals for locality aware loop scheduling schemes. The distinguishing characteristic of these schemes is that they are specifically designed for optimizing the behavior of data cache by assigning iterations to processor carefully. Such techniques either take advantage of the data reuse across different iterations (as in [8]) or exploit the fact that the same loop can be visited multiple times during the execution of the program [13]. Since optimizing cache locality means exploiting temporal and/or spatial reuse in the innermost loop iterations, one can expect a locality aware loop scheduling scheme to reduce bank energy consumption as well. This is the main reason why we compare, during the experimental evaluation, our approach to the locality aware loop scheduling schemes as well, in addition to pure static and dynamic scheduling techniques. Xue et al. [20] discuss a static scheduling scheme, specifically designed for software managed on-chip memories. There also exist several efforts that consider dynamic scheduling in the operating system (OS) [3] or system levels [21]. Our approach is different from such approaches in that in our case the compiler dictates the scheduling decisions, and we target embedded chip multiprocessors.

As far as the banked memory architecture is concerned, we are focusing on an SRAM based on-chip memory system, divided into multiple banks. In this system, each bank can be transitioned into a low leakage mode independently of the others to save energy. The specific policy we implemented in our simulations is based on [4], where the banks are placed into the low leakage mode periodically (using the suggested threshold values by [4]), and are activated when they are accessed (we assume the same reactivation penalties as in [4]). Consequently, working with a small set of banks at a given period of time increases the chances for the other (unused) banks to remain in the low leakage mode, thereby increasing energy savings. This is why our dynamic approach tries to schedule the loop iterations in such a fashion that the idle periods of the banks are lengthened as much as possible. Since we are not proposing a new bank power reduction strategy, due to space concerns, we do not discuss the related work on power saving in banked SRAM/DRAM memories; instead, we refer the reader to the papers [7, 10] and the references therein.

3. Bank Aware Loop Scheduling

As discussed in the previous section, in dynamic scheduling, when a worker CPU finishes its current work (assignment), it asks the master CPU to give it a new set of loop iterations. The important point to note is that the master has complete freedom in selecting this set of iterations, due to the fact that we consider parallelization of dependence-free loops only. In order to reduce the number of active banks at any given moment, our bank aware scheduler selects this set of iterations such that it *reuses* only the active banks, if it is possible to do so. Clearly, this complete bank reuse may not always be achievable, and when this is the case, our approach selects the iterations to be assigned to the requesting CPU such that the number of additional banks required is minimum. In mathematical terms, we represent the *current on/off status*[1] of the banked memory system using a bitmap ∇_c of the

[1] When there is no confusion, we use the terms such as low power mode, low leakage mode, and off mode interchangeably. It needs to be made clear however that, in our implementation, when a bank is placed into the low power mode, its contents are maintained, using an approach, similar to that discussed in [4].

following form:

$$\nabla_c = m_1 \bullet m_2 \bullet m_3 \cdots \bullet m_s,$$

where s is the number of banks in the memory system and m_i captures the status of bank i. Specifically, if m_i is 1, bank i is currently active, i.e., there is at least one processor in the system that executes an iteration which accesses that bank. On the other hand, $m_i = 0$ indicates that the bank is currently unused.[2] At a particular step, when the scheduler is about to assign a workload to a processor, it selects the set of iterations (that constitute the workload to be assigned) such that only the currently active banks are used if it is possible. Mathematically, let $\zeta(\vec{I})$ be a function that gives the set of banks that may be accessed by iteration $\vec{I}$ (Note that $\vec{I}$ is a vector where each entry corresponds to a value of a loop, starting from the outermost loop, corresponding to the first entry). Note that this is a *conservative* estimate since it may not always be possible to determine the exact set of banks to be accessed by a particular loop iteration. As a consequence, in the worst case, $\zeta(\vec{I})$ can contain all the banks in the memory system. Observe that a $\zeta(\vec{I})$ can also be represented as a bitmap $\beta(\vec{I})$ as follows:

$$\beta(\vec{I}) = n_1 \bullet n_2 \bullet n_3 \cdots \bullet n_s,$$

where n_i is set to 1 if bank i belongs to the $\zeta(\vec{I})$ set; otherwise, it is set to 0. Now, we can also define a workload level bitmap that captures the bitmaps of all the iterations in a given workload collectively. That is,

$$\tau(W) = k_1 \bullet k_2 \bullet k_3 \cdots \bullet k_s,$$

such that k_i is set to 1 if there is at least a $\vec{I} \in W$ such that n_i of $\beta(\vec{I})$ is 1; otherwise, k_i is set to 0. In other words, let $\vec{I_1}$, $\vec{I_2}$, $\cdots$, $\vec{I_l}$ be the set of iterations in the workload assigned to a processor when it asks for more work, and $\beta(\vec{I_j}) = n_{j,1} \bullet n_{j,2} \bullet n_{j,3} \cdots \bullet n_{j,s}$, where $1 \leq l$, Then, we have

$$k_i = n_{1,i} \vee n_{2,i} \vee n_{3,i} \vee \cdots \vee n_{l,i},$$

where $\vee$ denotes the OR operator.

The main job of our bank aware dynamic scheduler is to build a workload W at runtime and assign it to an idle processor which asks for more work to do. Let us first make the following definition. Given bitmaps p and q, we write $p \triangleright q$, if the following two conditions are satisfied together:

- The number of 1s in q is equal to or larger than that in p, and
- If p has 1 in ith position, q also has a 1 in its ith position.

For example, we have $11001000 \triangleright 11101001$ and $0101 \triangleright 0101$, while $1010 \triangleright 0101$ and $111000 \triangleright 101000$ are not correct. Based on this definition, the iterations to put in set W should be selected in such a fashion that $\tau(W) \triangleright \nabla_c$ should be satisfied if it is possible to do so. Before giving the pseudo code for the algorithm that selects W, the set of iterations to be assigned to a worker CPU that requires work, we want to discuss a couple of important issues.

First, it can be costly to compute a W set at runtime such that $\tau(W) \triangleright \nabla_c$ even if such a set that satisfies this condition does actually exist. Therefore, our approach does some extra work at compile time to reduce this potential runtime overhead. Let us use Z to denote the set of iterations to be executed for the current parallel loop (i.e., Z represents the iteration space of this loop). We divide at compile time this set into 2^s bins, where s is the number of banks in the memory system. Each bin holds the set of iterations that have the same $\beta(\vec{I})$ bitmap (in our implementation, each bin is represented by the constraints that give the iterations which belong to that bin). Clearly, some of these bins can be empty (i.e., there may not exist any iteration that accesses a particular set of banks). As the iterations are assigned to processors, we update the contents of these bins accordingly, a process during which some (originally full) bins can become empty. It is important to note that each bin can be represented using a bitmap, similar to those used for representing the on/off status of the banks (∇_c) and the bank access patterns of iterations ($\beta(\vec{I})$). Let $\mu(q)$ represent the bitmap of bin q. Suppose now that we are to assign a workload W (which contains K iterations to be selected from the iterations contained in Z) to a processor which asks for work. Using ∇_c, we first check the whether the bin q where $\mu(q) = \nabla_c$ has at least K iterations. If this is the case, we give K iterations to the requesting processor and we are done. If this is not the case (i.e., when bin q can provide only M ($< K$) iterations, we next search to find a set of bins $q_1, q_2, \cdots, q_r$ such that $\mu(q_1) \vee \mu(q_2) \vee \cdots \vee \mu(q_r) \triangleright \nabla_c$ and can collectively provide $K - M$ iterations for the requesting processor, where $M (< K)$ is the number of iterations provided by $\mu(q)$. Informally, this means selecting a set of bins such that collectively they provide $K-M$ iterations and these $K-M$ iterations do not demand access to any of the unused banks. Since there are many ways of selecting these banks and we cannot try all the alternatives at runtime due to cost considerations, we mark at compile time which alternative to try if we cannot find a bin q where $\mu(q) = \nabla_c$ has at least K iterations. In our current implementation, we try only one alternative which consists of two banks, and if that alternative cannot provide the required number of iterations, we select the remaining iterations randomly. It is also important to explain why we first try to find a bin q such that $\mu(q) = \nabla_c$ has at least K iterations. This is due to the following observation. At the time the slave processor in question asks for a workload, the set of banks that are active is captured by ∇_c. Since these banks are active anyway, we may want to reuse all of them (while they are active). This is because such a reuse will also likely to help cluster the iterations that do not use some subsets of these banks, which will in turn help increase power savings in the rest of the execution of this parallel loop nest. Figure 2 shows the compile-time portion of our approach based on the explanation above.

```
For i = 1, 2^s
  Generate the contents of bin q_i
  Compute, C(q_i), the size of q_i
  Select, if possible, two bins q_j and q_k such that μ(q_j) ∨ μ(q_k) = μ(q_i)
    and connect q_j and q_k to q_i
EndFor
```

Figure 2. Compile time component of our approach.

```
Obtain ∇_c
Determine the bin q_i to check
If C(q_i) ≥ K then
  Return the first K iterations in q_i to the requesting processor
  Update the contents of bin q_i
Else
  Take M iterations from q_i, where M < K
  Update the contents of bin q_i
  Determine bins q_j and q_k that are connected to bin q_i
  If C(q_j) + C(q_k) ≥ K − M then
    Return K − M iterations from these bins and update their contents
  Else
    Return the remaining iterations from any subset of bins and update contents
```

Figure 3. Runtime component of our approach.

Second, our bank aware loop scheduling approach can be used with any variant of self scheduling. This is because the different variants of self scheduling such as tapering [15] and factoring [6] differ only in the number of iterations they assign at runtime to a requesting CPU. Based on our discussion of the previous paragraph, this only affects the value of K (size of the workload W) and the rest of our approach can be used as it is. In our experimental evaluation however, we only implemented the bank aware version of the baseline self scheduling.

Third, it is important to understand why a purely static (bank aware) scheduling approach may not be as successful as our dynamic bank-aware scheduler. Notice that any purely static loop scheduling scheme that is to be bank aware needs to make conservation assumptions about the on/off status of the memory banks

[2] Such a bank can be in high leakage or low leakage mode, depending on how long it was unused.

	$\mu(q_i)$	Scenario I	Scenario II
q_1	1 0 0 0	8	7
q_2	0 1 0 0	8	27
q_3	1 0 1 0	20	2
q_4	0 0 1 0	5	6
q_5	0 1 1 0	5	4

Figure 4. Example application of our loop scheduling algorithm. The last two columns give the number of iterations in five bins; the other bins are assumed to be empty.

during each phase of the execution of the parallel loop. In other words, it needs to estimate the bitmap ∇_c conservatively (at compile time). However, such a conservative estimation can be far from reality due to at least two factors. First, dynamic cache behavior can affect the bank access pattern of a loop iteration completely. For example, for a given loop iteration, the compiler can conservatively deduce that it can access four banks; however, at runtime, all these four accesses can be captured and supplied by the data cache, resulting in no memory access. The second potential reason that can invalidate the compiler based estimation is the dynamic code behavior. For example, a procedure/function can have a lot of conditional branches. In order to make a conservative estimate, the compiler needs to consider the worst case scenario. However, in a given execution, only a subset of all possible branches can be taken, which means a much smaller number of bank accesses, compared to the conservative estimation made at compile time. Nevertheless, in our experiments, we also measured how much energy we would lose, had we adopted a static bank aware scheduling approach, instead of the dynamic approach presented in this paper. The pseudo-code for the algorithm that selects the workload W to be given to a requesting CPU is shown in Figure 3. This algorithm is executed at run-time and constitutes the dynamic portion of our approach (the static portion of our approach is given earlier in Figure 2).

We now give an example to illustrate how our approach works in practice using two scenarios presented in Figure 4. In this example, we assume that the memory system has four banks and at compile time we grouped the iterations into five different bins $(q_1, q_2, \cdots, q_5)$. Let us assume that the current bank on/off status, ∇_c, is 1010, the same as the bitmap of bin q_3. We further assume that the workload W we are to assign has 10 iterations. Note that each scenario in Figure 4 corresponds to a particular number of iterations at each bin. Under Scenario I, we first check q_3 to see whether it can supply the required number of iterations. Since this bin has currently 20 iterations and we need only 10 (i.e., K=10), we take the first 10 iterations from this bin, and reduce its contents to 10, and we are done. Under Scenario II, we also first check q_3. But, this time, this bin can supply only 2 iterations (i.e., M=2). So, we need $K - M$=10-2=8 more iterations. Assuming that q_1 and q_4 are identified as the backup bins for q_3 (since $\mu(q_1) \vee \mu(q_4) = \mu(q_3)$), we next check q_1 and q_4. Since q_1 can give us 7 iterations, we need only 1 (=10-(2+7)) iteration from q_4. After that, the contents of q_1 and q_4 are updated accordingly.

4. Evaluation

Using SIMICS [16], we simulated a chip multiprocessor architecture and evaluated the following loop scheduling schemes:

• **static:** This represents the well-known compiler based loop scheduling scheme. In this approach, the iterations of the loop to be executed in parallel are divided across the available parallel processors as evenly as possible. As noted earlier in the text, the main problem with this approach is that it cannot take into account the dynamic variances across the workloads of the different processors. In other words, distributing loop iterations evenly does not necessarily lead to evenly distributed workloads.

• **dynamic:** This is a well-known dynamic loop scheduling scheme (also known as self scheduling [19]). A master processor controls the loop distribution at runtime.

• **tapering:** This is a slight variant of the dynamic scheme, and we followed the specific implementation discussed in [15]. Our initial experiments that compared this scheduling scheme with trapezoid self-scheduling [18] and factoring [6] showed that all these three schemes exhibit similar behavior for our benchmark codes; therefore, we do not report separate results with the trapezoid self-scheduling or factoring.

• **locality aware-dynamic:** This is a dynamic locality aware scheduling scheme, explained in [8]. In this scheme, whenever a processor asks for a workload, it is given a set of loop iterations that exhibit high degree of data reuse among them. The goal of such an assignment is to improve the data cache locality. The reason that we make experiments with such a scheme as well is to demonstrate that a dynamic scheduling scheme that targets only cache locality may not be sufficient for maximizing bank energy savings.

• **locality aware-static:** This scheduling scheme is similar to the previous one, except that the assignment of loop iterations to processors are done statically (at compile time). Simply put, the iterations space of the parallel loop is divided at compile time into P subsets (P being the number of processors) such that the iterations in each subset reuse a lot of data elements among themselves. It generates similar results to the locality-aware static scheduling described in [13].

• **bank aware:** This is the scheduling scheme discussed in this paper (Section 3).

Parameter	Value
Number of Processors	8
IPC	2
L1 Cache (Per Processor)	8KB; 2-way; 32 byte line size
Shared On-Chip Memory	4MB; 8 banks of 512KB
L1 Access Latency	2 Cycles
On-Chip Memory Access Latency	8 Cycles
Bus Contention Cost	5 Cycles

Table 1. Default values of our simulation parameters.

Benchmark Name	Explanation	Dataset Size	# of Cycles	Energy Consump
Baleen	Segments an image into subimages	2.85MB	344.52M	171.48mJ
Demosaic	Interpolates a complete image from partial raw data	3.51MB	576.12M	247.18mJ
Imar_ver2	Transforms different sets of data into one coordinate system	2.51MB	305.83M	148.17mJ
Zonography	A variant of linear tomography kernel	3.94MB	883.9M	481.92mJ
Poly 1.1	A complex form of tomography (poly tomography)	3.98MB	927.06M	515.64mJ
Cbd	Car barrier detection algorithm	1.73MB	290.46M	129.13mJ

Table 2. Benchmark codes used in our experiments. The numbers under the last two columns are for the static loop scheduling scheme. The energy numbers are calculated for 70 nm.

The code modifications required by these schemes are automated using the SUIF infrastructure [5]. As mentioned earlier, we used the SIMICS [16] platform to perform our experimental evaluation. SIMICS is a functional simulator and runs unmodified operating systems, drivers, firmware, and application software on the simulated machines. As far as the software is concerned, there is no difference from a real machine. We used this platform to simulate an embedded chip multiprocessor system with private (on-chip) L1 and shared (on-chip) SRAM memory, which is banked. The default number of banks is 8 and each bank is 512KB (meaning that the total on-chip memory space is 4MB). The architecture has separate L1 instruction and data caches for each and every processor. The default simulation parameters used in most of our experiments are given in Table 1.

Table 2 present the important characteristics of the benchmarks used for evaluating our bank aware dynamic loop scheduling scheme. The third column of this table gives amount of data manipulated by each benchmark, and the fourth column shows the execution cycles taken by a pure static scheduling scheme (as explained above). The last column of the table gives the energy consumption in the memory system, again under the static loop

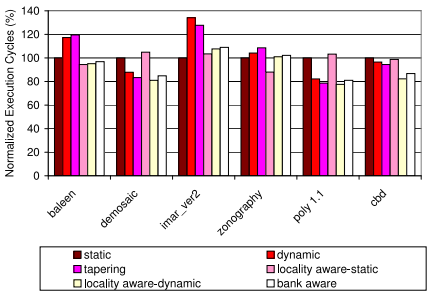

Figure 5. Execution cycles normalized with respect to the static scheduling scheme.

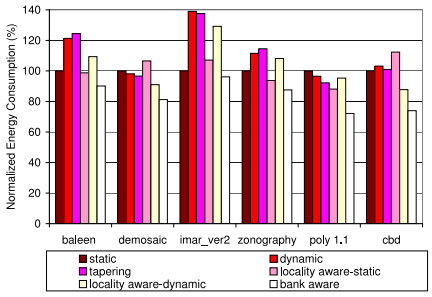

Figure 6. Energy consumptions normalized with respect to the static scheduling scheme.

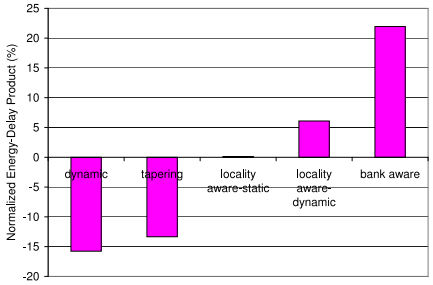

Figure 7. Average energy-delay products normalized with respect to the static scheduling scheme.

scheduling scheme. The performance and energy numbers presented in the rest of this section are given as values, *normalized* with respect to the last two columns of this table.

Figure 5 presents the execution cycle results. One can easily see that the dynamic scheduling scheme (second bar) generates savings (over the static scheme) in three applications, namely demosaic, poly 1.1, and cbd. These are exactly the benchmarks with large workload variations across processors when the static workload assignment is employed. In the other three benchmarks, however, the dynamic scheme generates poor results. Overall, as compared to the static scheme, the dynamic scheme ends up with 3.7% performance degradation when averaged over all six benchmark codes. The behavior exhibited by the tapering scheme (third bar) is similar to that of the dynamic scheme, with an average performance degradation of about 2% over the static scheme. In comparison, locality aware-static scheme (fourth bar) performs better, bringing reasonable improvements in two benchmarks (baleen and zonography). The locality aware-dynamic scheme (fifth bar) is much more successful since it is able to both take advantage of data reuse and exploit load imbalances. The locality aware static and dynamic schemes bring average performance improvements of 1.2% and 9.2%, respectively, over the static loop scheduling scheme. Lastly, our bank-aware loop scheduling scheme (last bar) achieves 6.5% improvement over the static scheme. Although it is not as good as the locality aware-dynamic scheme (as the latter is pure performance oriented), it is not too far from it either. This is because of the fact that, most of the time, minimizing the number of accesses to a small set of banks (which is the main goal of our approach) also leads to good data cache behavior, as it tends to improve data reuse within a given time period.

The normalized energy consumption results are presented in Figure 6. The energy consumptions for on-chip memory components (L1 cache and on-chip memory) are calculated with the help of the CACTI toolset [2]. The energy consumption of the remaining components on the other hand are obtained using activity based energy models similar to those used in Wattch [1]. Maybe the most important observation one can make from the results in Figure 6 (which include energy consumption of both memory and non-memory components) is that the bank aware scheme performs much better than the remaining schemes, bringing an average energy saving of 16.4% over the static loop scheduling scheme. In fact, it reduces energy in all six benchmark codes. In comparison, the remaining scheduling schemes do not repeat the savings they achieve in execution cycles. For example, the dynamic, tapering, locality aware-static, and locality aware-dynamic scheduling schemes increase the energy consumption of the static scheduling scheme by 11.6%, 11.1%, 1,1%, and 3.4%, respectively, on average.

Since any loop scheduling scheme affects both execution cycle count and energy consumption, it is also important to quantify the energy-delay product values. Figure 7 gives the energy-delay products when *averaged* over all six application codes. Each bar in this figure is normalized with respect to the average energy-delay product of the static loop scheduling scheme. Since the bank aware scheduling approach improves both performance and energy consumption, it exhibits the best energy-delay product. We also see that, as well as energy-delay product is concerned, the locality aware-dynamic scheme is the only scheme (other than our bank aware approach) that brings some reasonable improvement.

Figure 8 presents the *average* performance and energy values with the different processor counts (4, 8, 12, and 16). Remember that our default processor count was 8. We see that, as we increase the number of processors, the differences among the different scheduling schemes get magnified (this is true for both performance and energy). The main reason for this is the fact that an increase in the processor count usually leads to spread the bank accesses more in the memory space (i.e., more irregularity in the bank accesses). Considering the possibility that future chip multiprocessors will accommodate large number of CPUs, we believe that these results are encouraging. Figure 9 gives the energy results with different number of banks, keeping the total memory space at 4MB. All other parameters are set to their default values shown in Table 1. We see that the behavior of the locality aware scheduling schemes improve with smaller number of banks. This is because as the number of banks gets smaller, optimizing for cache locality generates similar results to those obtained by optimizing bank locality. However, when the number of banks is increased, these two locality concepts start to behave differently, and as a consequence, our bank aware scheduling scheme generates much better results than the others.

We next study an alternate bank aware scheduling scheme, which is a *static version* of the dynamic scheme discussed in this paper. The only difference between this scheme and our dynamic scheme is how the ∇_c bitmap is obtained. In our dynamic scheme, it is obtained at run-time, while in this alternate bank aware scheme, it is computed at compile time. As discussed earlier in Section 3, the compile time computation of ∇_c can be overly pessimistic. The energy results captured by the first two bars (for each benchmark) in Figure 10 corroborate this expectation. We see that the average energy improvements by the dynamic and static scheduling schemes are 16.4% and 10.4%, respectively (the results for the dynamic scheduling scheme are reproduced from Figure 6). These results underline the importance of dynamically obtaining the ∇_c bitmap. To better understand the difference between the statically computed and dynamically obtained bitmaps, we also recorded during the experiments, the causes for misprediction (of the ∇_c bitmap) with the static scheme. The results are presented in Figure 11. As discussed earlier, cache behavior and dynamic control flow are two important reasons for the conservative estimation of the bitmap, also corroborated by the results in this graph. The third portion of each bar in this graph represents the mispredictions whose cause we could not identify.

Recall that in our current implementation we make two attempts to select the set of iterations that satisfy the requirement that no new banks are activated by the newly-assigned workload. In the first attempt, we try find a bin q where $\mu(q) = \nabla_c$ has at least K iterations. If this fails, in the second attempt, we try to find a set of bins q_1 and q_2 such that $\mu(q_1) \vee \mu(q_2) \triangleright \nabla_c$ and can collectively provide the $K - M$ iterations, where M is the number of iterations provided by bin q. If this try also fails, then we

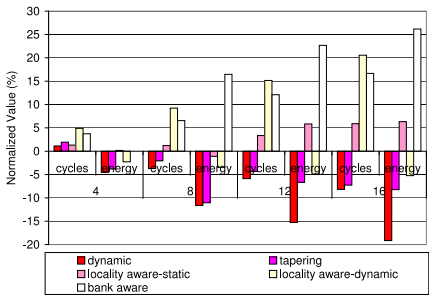

Figure 8. Average energy consumption and execution cycle results with the different processor counts.

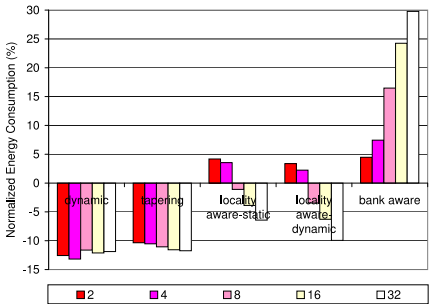

Figure 9. Average energy and execution cycle results with the different bank counts. In each experiment, the total memory capacity is the same.

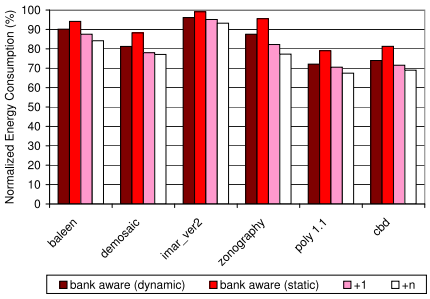

Figure 10. Energy comparison of different schemes.

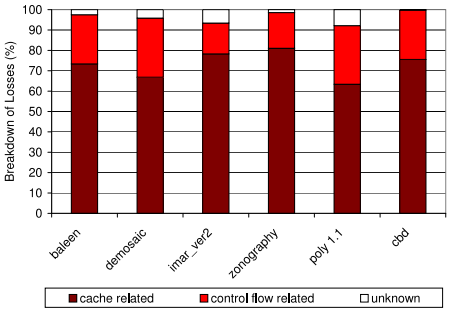

Figure 11. Breakdown of causes for mispredictions of the ∇_c bitmap when the static bank aware scheme is used.

select the remaining iterations required randomly. However, it is clear that, further improvements to this implementation are possible by increasing the number of attempts. In the general case, when the first attempt fails, we can search for a set of bins q_1, q_2, $\cdots$, q_r such that $\mu(q_1) \vee \mu(q_2) \vee \cdots \vee \mu(q_r) \triangleright \nabla_c$ and these bins can collectively provide the required number of iterations. Note that in general there are many ways of selecting these bins. The last two bars for each benchmark in Figure 10 represent the energy consumption values with two enhanced implementations of our bank aware dynamic loop scheduling scheme. The bar marked using "+1" tries one more alternative over our second attempt in the default implementation, whereas the bar marked using "+n" tries all possible alternatives. We see that the average energy savings brought by the "+1" scheme and "+n" scheme are 19.1% and 21.9%, respectively. Considering these values with our default value (16.4%), we can conclude that our default implementation is not far from them, as far as energy consumption is concerned. In addition, although not presented here in detail, the "+n" scheme increased the execution cycles by nearly 6% over our default implementation, making the latter even more promising option, when energy consumption and execution cycles are considered together.

5. Conclusions and Future Work

The main contribution of this paper is a memory bank-aware dynamic loop scheduling scheme. Our approach selects the set of iterations to assign to a requesting processor such that the currently active banks are reused if possible (without activating a new bank). We tested this approach and collected both performance and energy numbers using a SIMICS based simulation platform. In our evaluation, we also compared it to a number of previously published loop scheduling schemes, including the pure static and dynamic schemes, variants of dynamic scheme, as well as two locality oriented loop scheduling approaches. Our experimental results with six embedded applications clearly show that the proposed scheduling scheme not only reduces the energy consumption significantly, but it also leads to much better energy savings when compared to these prior techniques and it is competitive with the loop scheduler that generates the best performance. We are currently in the process of applying this scheduling scheme to cluster based chip multiprocessor systems.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*. June, 2000.

[2] *CACTI Toolset.* `http://www.research.compaq.com/wrl/people/jouppi/CACTI.html`

[3] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* 1994.

[4] K. Flautner, N. Kim, S. Martin, D. Blaauw, T. Mudge. *Drowsy Caches: Simple Techniques for Reducing Leakage Power.* In Proceedings of the 29th International Symposium on Computer Architecture, Anchorage Alaska, May 2002.

[5] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December, 1996.

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

[7] M. Kandemir, I. Kolcu, and I. Kadayif. Influence of Loop Optimizations on Energy Consumption of Multi-Bank Memory Systems. In *Proc. of International Conference on Compiler Construction*, Apr 2002.

[8] M. Kandemir. LODS: Locality-Oriented Dynamic Scheduling for On-Chip Multiprocessors. In *Proceedings of the 41st Design Automation Conference*, San Diego, CA, June 2004.

[9] M. S. Lam. Locality Optimizations for Parallel Machines. Parallel Processing: CONPAR'94 - VAPP VI. *Third Joint International Conference on Vector and Parallel Processing,* September 1994.

[10] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power-Aware Page Allocation. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov 2000.

[11] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Processing,* 1993.

[12] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, June 1992.

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

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

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

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

[17] P. Tang and P.-C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loops. In *Proceedings of International Conference on Parallel Processing,* August 1986.

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

[19] M. Wolfe. *High Performance Compilers for Parallel Computing,* Addison-Wesley Publishing Company, 1996.

[20] L. Xue, M. Kandemir, G. Chen, and T. Yemliha. SPM-Conscious Loop Scheduling for Embedded Chip Multiprocessors. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, July 2006.

[21] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SoCs. *IEEE Design and Test of Computers,* 2001.