Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems*

V. De La Luz
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
delaluzp@cse.psu.edu

M. Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK
i.kolcu@co.umist.ac.uk

ABSTRACT

An architectural solution to reducing memory energy consumption is to adopt a multi-bank memory system instead of a monolithic (single-bank) memory system. Some recent multi-bank memory architectures help reduce memory energy by allowing an unused bank to be placed into a low-power operating mode. This paper describes an automatic data migration strategy which dynamically places the arrays with temporal affinity into the same set of banks. This strategy increases the number of banks which can be put into low-power modes and allows the use of more aggressive energy-saving modes. Experiments using several array-dominated applications show the usefulness of data migration and indicate that large energy savings can be achieved with low overhead.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures

General Terms

Design, Experimentation, Performance

Keywords

Energy Consumption, Multi-Bank Memories, Data Migration

1. INTRODUCTION

The high energy consumption of main memories make them target of many energy-conscious optimization techniques. On current computer systems, main memory is one of the energy bottlenecks of the architecture [2]. This is especially true for mobile applications which are typically memory-intensive (array-dominant such as signal and video processing). In addition, embedded applications are gradually becoming more data-centric with stringent memory requirements (both for storage and speed), causing vendors to incorporate large storage capacities into their offerings. An architectural solution to this problem is to adopt a multi-bank memory system instead of a monolithic (single-bank) memory architecture. In a multi-bank memory architecture, unused memory banks (idle banks) can be disabled, thereby saving energy. However, in many circumstances, disabling a memory bank completely would not be possible because banks usually hold some useful data (even if they are not frequently accessed). Rather, some recent memory systems (e.g., [10]) provide low-power operating modes which enable large energy savings without completely disabling memory banks. More specifically, a bank that is not currently servicing a memory request can be put in a low-power operating mode. A major problem with these energy-saving modes is that they incur additional delay (called resynchronization cost or resynchronization penalty) when a bank in one of these modes needs to service a memory request. The magnitude of this delay (performance penalty) depends on the specific power mode; typically, a more energy-saving mode also incurs more delay. Consequently, it is advantageous to keep memory banks in idle conditions for long durations of time so that the resynchronization penalties can be compensated.

Our goal in this paper is to reduce the overall energy consumption of a given application by reducing the contribution of the component due to a multi-bank main memory with low-power operating modes. One way of achieving this is to cluster data accesses in a small number of memory banks and place the remaining banks (idle banks) into a low-power operating mode. In many cases, the default access pattern may not lead to clustered accesses and intrinsic dependences between data accesses and those due to control flow prevent the most suitable computation transformation from clustering the accesses. An alternative approach is to modify the data layout in memory such that the data elements that are accessed contemporaneously (i.e., those with temporal affinity) are stored in a small set of banks. As in the computation transformation case, such data transformations enable us to place a large number of (idle) memory banks into a low-power operation mode.

Previous compiler-oriented work has focused on implementing static data transformations [6] using which the array layouts in memory are transformed and these transformed layouts remain in effect throughout the execution of the code. Among the data transformations investigated are array interleaving and array co-location. In the operating system (OS) area, Lebeck et al. [7] has studied the OS support for exploiting low-power operating modes. In many applications, the data access patterns are not fixed during the entire execution and vary dynamically as the program executes. In an array-dominated embedded application, different arrays might be used together in different parts of the code. So, there is a need for a dynamic data co-location scheme taking into account temporal affinity (proximity) between data structures.

In this paper, we present a dynamic data migration scheme which co-locates the data elements automatically (as the program executes) to take advantage of the variations in inter-array access patterns. Our approach is based on a runtime library called Data Mi-

*This work is supported in part by NSF Career Award #0093082.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA

grator (DM) which samples the arrays that are accessed together and keeps temporal access relations between arrays using a table. From time to time, this table is scanned and the arrays with high number of contemporaneous accesses are determined. These arrays are then dynamically migrated across memory banks to ensure that they reside in a small set of banks. The proposed runtime system has been implemented and a simulation framework to evaluate its effectiveness has been built. Our results obtained using applications that manipulate large datasets residing in multi-bank memory configurations show that dynamic data migration improves energy behavior by as much as 40%.

Next section gives a brief review of a multi-bank memory architecture and explains low-power operating modes. Section 3 presents the important characteristics of our data migration scheme. Section 4 gives a description of our implementation (Data Migrator) and illustrates how it interacts with application execution. Section 5 describes our experimental platform, gives information about the codes executed, and reports experimental data showing the usefulness of dynamic data migration. Section 6 presents a summary of the paper.

2. MEMORY ARCHITECTURE

In this work, we assume a multi-bank/multi-module memory system. Each bank can contain multiple modules and is assumed to be a single word wide. Modules/banks can be placed into low-power operating modes independently. One example of this type of memory architecture is RDRAM [10] in which each bank contains a single module and can work with four operating modes: active, standby, nap, and power-down. A bank can service a read/write request only in the active mode. If it is not servicing a memory request, it can be placed into one of low-power operating modes (standby, nap, or power-down). In this paper, we assume a single module per bank; consequently, we use the terms module and bank interchangeably.

Each low-power mode is implemented by disabling specific parts of the DRAM chip and can be described using two metrics: energy consumption and resynchronization cost. The first metric is the per access (or per cycle) energy consumption when the module is in a specific mode whereas the second metric is the time (in cycles) it takes to bring a module back to the active mode (i.e., performance penalty). Typically, lower the energy consumption, higher the resynchronization cost. Figure 1 shows the energy consumptions (per access) and resynchronization costs for the operating modes used in this study. Obviously, there is a tradeoff between energy saving and performance penalty when selecting a low-power mode to use for a given idle bank. The time between successive accesses to a given bank (called inter-access time) is the major determining factor in selecting the most suitable low-power operating mode. The inter-access time can be estimated either by an optimizing compiler and can be predicted by a hardware/runtime mechanism. Both of these strategies have been investigated in a previous paper [6]. In this study, we adopt a hardware-based strategy in which the bank inter-access times are predicted using a prediction logic attached to memory controller. The rationale behind the hardware-based predictor is that if a memory bank has not been accessed in a while, then it is not likely to be needed in the near future (that is, inter-access times are predicted to be long). A threshold is used to determine the idleness of a bank after which it is transitioned to a low-power mode. More specifically, in our current implementation, after 10 cycles of idleness, the corresponding bank is put in standby mode. Subsequently, if the bank is not referenced for another 100 cycles, it is transitioned into the nap mode. Finally, if the bank is not referenced for a further 1,000,000 cycles, it is put into power-down mode. Whenever the bank is referenced, it is brought back into the active mode incurring the corresponding resynchronization costs (based on what mode it was in). Implementation details of this hardware-based scheme are beyond the scope of this paper and can be found in [6].

Operating Mode	Energy Consumption (nJ)	Resynchronization Cost (cycles)
Active	3.570	0
Standby	0.830	2
Nap	0.320	30
Power-Down	0.005	9,000

Figure 1: Energy consumptions (per access) and resynchronizations costs for our operating modes. During transitions from a low-power mode to the active mode, a full active mode energy is assumed to be spent.

3. DATA MIGRATION FOR LOW POWER

Automatic data migration is beneficial when a poor initial placement of the data on memory banks is responsible for large energy consumption and the techniques based on computation restructuring are not able to improve the energy behavior of the application. The idea behind data migration is to bring together the datasets that have originally been placed into different banks but have frequently been accessed together lately. Such datasets (e.g., arrays) are said to exhibit temporal affinity. The expectation is that if two (or more) datasets have been accessed together in the past, they will continue to be accessed together in the near future. By bringing the arrays with temporal affinity together, we increase the chances that fewer number of banks need to be active in a given period of time. In this paper, we focus on array-dominated/loop nest based codes which can be found in domains such as image/video processing and scientific computing. This paper reports experimental results for a single program environment, and does not consider the existence of a virtual memory system (i.e., the data migrator deals directly with physical addresses). There are several important issues that need to be addressed in a data migration scheme for low power. Below we discuss these issues and explain the rationale behind our design and implementation choices.

3.1 Granularity of Migration

The migration granularity might be an entire array or an array region (section). While a fine-granular migration scheme has the potential of localizing the accessed data elements into smaller number of banks better (that is, exploiting a finer-granular temporal affinity), it also entails dividing a given array across banks in a noncontiguous manner. That is, different portions of an array can be stored in different memory banks and sections of different arrays can be interleaved. This, in turn, makes addressing the array in question more difficult (as we work with physical addresses). Consequently, in this work, we chose to migrate entire arrays (when migration is beneficial).

3.2 Detection of Temporal Affinity

To place data elements that are likely to be accessed contemporaneously in the same memory bank, we need to capture which data elements have recently been accessed together. This can be done by sampling the data accesses at runtime (dynamically). Obviously, cycle-by-cycle sampling would affect performance negatively; so, a strategy might be adopting a sampling threshold. More specifically, at regular intervals (sampling threshold), the data accesses are sampled, and the information collected is used to update a table called affinity table. Each entry in the affinity table is a triplet of the form $(U_i, U_j, count_{ij})$ which indicates that arrays U_i and U_j have been accessed together $count_{ij}$ times (up to the current point in the execution). If the current sampling indicates that U_i and U_j have been accessed together (since the last sampling), there are two possibilities. If we do not have any entry (triplet) in the affinity table for this relation between U_i and U_j, we create such an entry, set its count value to 1, and insert it in the affinity table. If, on the other hand, such an entry already exists in the table, we simply increment the corresponding count value. In cases where the sam-

pling indicates that there are $k > 2$ arrays, U_1, U_2, U_3,...,U_k, with temporal affinity, we update all entries $(U_i, U_j, count_{ij})$ where $1 \leq i < j \leq k$ in the table (or create these entries if they are not already there). The size of the affinity table is another issue that needs to be considered. In many data-intensive image/video codes, the number of arrays are limited; this also puts a limit to the table size. Nevertheless, if the size of the table is expected to be large, one might also consider using a small table and managing it as a fully-associative cache. In this case, if, during sampling, we detect a new temporal relation and the table is full, we need to discard a table entry. Possible choices here are to select the entry with the minimum count value or to select the least recently accessed entry.

3.3 Tracing Temporal Affinity

The count values in table entries should be decremented from time to time to adapt the counts to the most recent locality. This is mainly to prevent an old temporal affinity relation from dominating the current (active) affinity relations. For example, imagine two arrays U_i and U_j that have been used together in the past and led to the accumulation of a large $count_{ij}$ value. It might happen that if this count value is never decreased it can still be considered as the dominant count even after a long time this temporal affinity relation has died. One approach would be adopting a threshold, a regular interval, using which the count values are decremented periodically. This, however, involves an extra overhead from the performance viewpoint. A better scheme would be decrementing count values at sampling points. That is, when we increase the count values of the currently accessed arrays (or when we create new entries in the table), we can also decrement the count values of all other (not updated) entries in the table. Such a strategy obviously rewards the most recent temporal affinity relations and punishes the relations that have not been active most recently. In our current implementation, we adopt this strategy.

3.4 Initiating Migration

Another important issue is to decide when the migration should start. Typically, we can select a migration period (migration threshold), a regular interval, at which the affinity table is checked to search for the potential arrays to be migrated. Selecting a suitable migration threshold for a given application and its access pattern is a critical issue. A very small threshold may incur a large overhead, whereas a very large threshold may miss important opportunities for bringing the arrays that have most recently been accessed together. It is not easy to determine a suitable migration threshold without extensive experimentation. Consequently, we experimented with different migration threshold values for each benchmark code we considered. As explained above, the affinity table maintains the temporal affinity relations between array pairs. Independent of how frequently the entries in this table are checked, we need to have a strategy for determining the arrays to be migrated. This is explained next.

3.5 Determining the Arrays to be Migrated

Given an affinity table, we can use different strategies to select the arrays to be migrated. Different strategies have different overhead and performance tradeoffs. In this study, we mainly consider a class of strategies called the first m triplet (mFT) strategies. A 1FT strategy looks at (when the migration period arrives) only the triplet with the highest count value and tries to bring the involved arrays together as much as possible. Similarly, a 2FT strategy looks at the two triplets with the two highest count values, and so on. As we increase the number of triplets to consider, we can expect that it is going to take longer time to decide an appropriate migration scheme (i.e., how the arrays should actually be migrated) as more triplets will usually involve more arrays. A straightforward implementation of an mFT strategy just determines the entries with the highest m count values, picks up the arrays names, and tries to implement a migration strategy as explained below such that the arrays whose

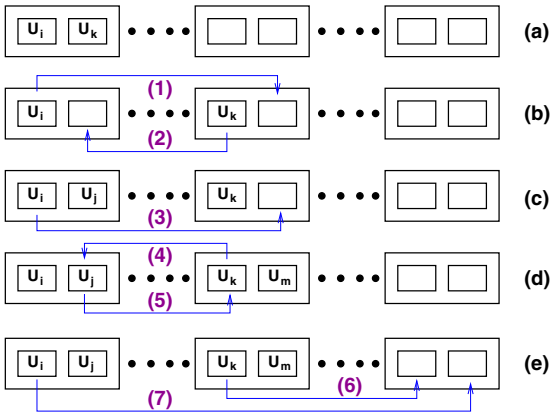

Figure 2: Five different migration scenarios that involve two arrays (U_i and U_k). For clarity, only three banks are explicitly shown.

names appear in each selected triplet are brought together in a single bank (or in the minimum number of banks). In case of conflicts (i.e., different triplets demand different migrations for the same array), we can give priority to the triplet with the higher count value. An alternative class of strategy, which we call the limit-n (nLM) strategy considers a triplet if and only if it's count value is larger than or equal to n. Note that it is also possible to adopt a hybrid strategy by combining the mFT and nLM strategies. For example, a conservative strategy can consider a triplet (that is, the arrays in the triplet for migration) if and only if its count value is larger than n and this value is among the highest m count values.

3.6 Scheduling Movements

Once the arrays to be migrated are selected, a strategy should be decided upon using which the array transfers (migrations) between memory banks are scheduled. The complexity of the schedule depends on the number of arrays involved in the migration. Let us first give a description of the scheduling algorithm when only a single triplet (i.e., two arrays) is involved. Later, we explain how this algorithm can be extended to work when more than two arrays are involved. Figure 2 illustrates five different scenarios assuming that we would like to honor the triplet (U_i,U_k,$count_{ik}$); that is, we want to bring arrays U_i and U_k together. If these two arrays already reside in the same bank as shown in Figure 2(a), nothing needs to be done. In Figure 2(b), on the other hand, these two arrays are in different banks and there is sufficient space in each bank for the other array. In this case, we have two options. We can either migrate U_i to the bank that holds U_k, or vice versa. These two actions are marked using (1) and (2) in Figure 2(b). Figure 2(c) depicts a scenario similar to that in Figure 2(b) except that the bank that holds U_i does not have sufficient space for U_k. Consequently, we have only one option: migrating U_i to the bank that holds U_k. This action is marked (3) in the figure. In Figure 2(d), both the banks that hold the arrays are full. In this case, we need to find an array (or arrays) in the bank that holds U_i and interchange the locations (banks) of this array and U_k; or alternately, we need to find an array (or arrays) in the bank that holds U_k and interchange the locations of this array and U_i. Figure 2(d) depicts the former case and the necessary migration activities are marked using (4) and (5). In cases where we may not be able to find a suitable array to interchange with U_i or U_k (as, for example, such an interchange will distort another important temporal affinity relation), we might want to consider to migrate U_i and U_k, together, to another memory bank as depicted in Figure 2(e). In the figure, these migration activities are marked using (6) and (7). Obviously, it is a good idea to select a bank that already contains some arrays instead of a new bank which would

have otherwise been completely disabled (consuming no power). Given two arrays that need to be brought together, our current implementation checks the feasibility of the options in Figure 2 in the order given, and selects the first option that applies to the case at hand. Note that, in general, our strategy for migration tries to minimize the number of migrations. In cases where we have multiple alternatives (to achieve the same result) as in Figure 2(b), we take into account the sizes of the arrays as well; that is, everything else being equal, it is preferable to migrate the smaller array to the bank that holds the larger one.

When more than two arrays are involved in the migration (e.g., as a result of taking into account more than one table entry), scheduling migrations becomes a more challenging problem. Although not shown here due to lack of space, it can be proved that optimal migration problem that involves multiple table entries is NP-hard. Therefore, a polynomial-time solution is unlikely to exist. Instead, we propose here a heuristic solution that works well in practice. In the first step of our heuristic, we (logically) cluster all the involved arrays into groups. Each group contains the arrays that are expected to be in the same (set of) bank(s) for the best memory energy behavior. As an example, consider a scenario where we consider three triplets simultaneously: (U_1,U_2,$count_{12}$), (U_3,U_4,$count_{34}$), and (U_2,U_5,$count_{25}$). Here, the first group contains arrays U_1, U_2, and U_5 whereas the second group contains arrays U_3 and U_4. In the second step, we rank the groups according to their relative importance. This ranking helps us later in resolving the conflicts if different groups demand different (conflicting) migrations. To rank the groups, we use the concept of group weight (denoted gv_k for a given group k) which is simply the sum of the counts of the triplets that supply the arrays in the group. In our current example, gv_1 is $count_{12}$ + $count_{25}$ and gv_2 is $count_{34}$. In the third step, we start with the the group with the highest weight and try to co-locate the arrays in this group as much as possible using minimum number of migrations. Our current implementation achieves this starting with the pair (of arrays) with the highest temporal affinity and tries to bring these arrays together. It then moves to the second pair (of highest affinity) and tries to satisfy this relation, and so on. Note that a given array in a group can be involved in multiple affinity relations. The approach then moves to the second group (i.e., the one with the second highest group weight), and in migrating it, it tries not to re-locate any of the arrays which have been involved in the previous migration (i.e., the one that involves the arrays belonging to the group with the highest weight). The algorithm continues in this fashion until all the groups have been processed.

3.7 Backup Policies

We need backup policies for at least two reasons. First, it might happen that (as a result of our array selection strategy such as nLM) we may not be able to find a triplet in the affinity table to use for migration. In this case, a backup strategy could be just not performing migration at this point (a low-overhead solution). Another backup strategy could be relaxing the strategy that we use to select the triplet and re-probing the affinity table. Although this latter alternative will have a larger overhead (as compared to the former), we can expect that it will perform better. Second, we need a backup policy when the arrays that we want to co-locate already reside in the same bank(s). As in the previous case, we can adopt here at least two alternatives. We can either continue with the execution and drop out migration at this point, or we can re-probe the affinity table to select alternative arrays for migration. Note, however, that in this last case, the migration decisions to be taken should not cause the arrays that have been determined (during the first probe, before re-probing) to be co-located but have been found to be residing on the same bank(s) to be separated.

3.8 Data Freezing and Defrosting

Arrays that exhibit different access patterns in different parts of the computation may be repeatedly migrated from one memory bank to another. Such frequent data migrations can easily offset potential energy gains from optimal data placements in banks. To avoid these costly frequent migrations, it is possible to freeze these arrays in specific banks (also called pinning in this paper) after a certain number of migrations. These frozen arrays can be defrosted after some time so that they become eligible again for migration. To decide whether an array should be pinned or not, we associate a migration counter with each array (initialized to zero). Each time an array is migrated this counter is incremented by one. When it reaches a certain value (migration limit), we can freeze the array on the bank where it currently resides. The migration counters can be decremented at regular intervals (defrosting threshold). To reduce overhead, this interval might be a multiple of the sampling threshold.

3.9 Overheads

There are two major sources of overhead in our migration scheme. The first is that the overall time spent by the program in the migration-related activities. These activities include sampling for temporal affinity, determining the arrays to be migrated, the migration overhead (time) itself, and updating counters (e.g., those keep track of the number of migrations, etc.). This cost is highly implementation dependent and can be made a small proportion of overall execution time in a careful implementation. Second, data migration consumes some amount of energy during copying data from one bank to another. Again, this cost can be decreased by reducing the number of arrays to be migrated, by using fast memory-to-memory copies (e.g., block copies), and by not migrating arrays unless it is really beneficial to do so.

4. DATA MIGRATOR (DM)

The ideas presented in the previous section have been implemented in a software package called Data Migrator (DM). The DM is designed as a runtime library which contains a set of functions and runs as a single thread of control when activated. The programmer initializes the DM by supplying the values for a set of parameters (to a *DM_initialize* routine). After that, the user activates the library by calling a special *DM_activate* routine. From this point on, the DM runs (as a separate thread) parallel with the application. It interrupts the application at regular intervals which are determined by the parameters supplied in the *DM_initialize* routine (more specifically, by the sampling threshold).

For ease of implementation, all thresholds are set to multiples of the sampling threshold which is used to interrupt the execution of the application. At each interrupt, the DM code is executed. A sketch of this code is given in Figure 3. Once activated, the DM thread checks for the occurrence of three events (using the three outermost 'if' statements shown in Figure 3). The first statement is always true as the DM thread is invoked when sampling threshold arrives. The thread then visits the affinity table and updates it (i.e., increments the count values for the triplets whose arrays pairs are accessed and decrements the count values for all other triplets). After that, the DM thread checks whether the migration threshold has arrived. If it has, the thread runs the migration policy (e.g., mFT, nLM, or a hybrid policy). As a result of this policy, we might have arrays that need to be migrated. If such arrays exist, we migrate them as explained in the previous section; and if not, we run the backup policy (and migrate arrays if required). In any case, we need to update the migration counter and freeze the array(s) if the counter overflows. Finally, we need to check whether the defrost threshold has arrived, and if so, make the frozen arrays migratable.

5. EXPERIMENTS

This section explores the overall energy improvement due to array migration and quantifies its energy/performance overheads. Figure 4 gives important characteristics of the array-dominated codes

```
if (sampling threshold arrives) then
 update affinity table;
if (migration threshold arrives) then
 {
  run migration policy;
  if (there are arrays to be migrated) then
   migrate arrays;
  else run backup policy;
  for each array involved in migration do
   update migration counter;
   if (migration counter overflows) then
    freeze the array;
 }
if (defrost threshold arrives) then
 initialize migration counters for all frozen arrays;
```

Figure 3: The DM thread.

Benchmark	Source	Input Size (MB)	Base (mJ)	Overhead	ST	MT
adi	Livermore	19.2	1,492	2.5%	1	100
apsi	Perfect Club	42.4	499	16.2%	1	100
btrix	Spec	48.8	719	7.6%	1	100
eflux	Perfect Club	33.5	98,501	4.8%	1	500
phods	-	32.9	37,785	2.8%	1	1,000
tomcatv	Spec	28.0	19,850	7.7%	1	100
vpenta	Spec	44.0	13,714	6.8%	1	1,000

Figure 4: Benchmark codes used in the experiments and their important characteristics.

that we used in our experiments. We measured the benefits of dynamic data migration using a set of seven array-dominated codes. `phods` is a motion estimation code whereas other codes are from array-based benchmarks. The third column gives the total input size (in MBs), and the fourth column gives the memory system energy consumption (in millijoules) for the base case (see below) using four 16MB memory banks (i.e., a total of 64 MB off-chip memory). The memory system energy includes the energy consumed in DRAM banks as well as the energy spent in 16KB, 2-way set-associative data cache with a line size of 32 bytes. The fifth column reports the percentage increase in the execution time of applications due to migration (i.e., performance overhead). We see that except for one case, the performance overhead of array migration is bounded by 8%. Finally, the last two columns give the sampling thresholds (denoted ST) and migration thresholds (denoted MT), respectively, used in majority of our experiments (in units of thousand cycles). In all experiments, we set the back-up policy to do nothing as we have found that this policy generated comparable results to more aggressive second choice-based policies without incurring as much overhead. Also, all the energy figures reported here include the energy overhead of data migration. We also used a powerful back-end compiler which performs all major classical low-level optimizations such as instruction scheduling and global register allocation [9].

5.1 Energy Benefits

Figure 5 presents the normalized energy consumptions when dynamic data migration (the 1FT policy) is employed with respect to a base case which does not use migration but tries to determine the best possible static data layout (array mapping) using the technique discussed in [6]. To evaluate the energy impact of array migration, we experimented with two initial data mappings. In the first mapping, which we call sequential mapping, the arrays are stored in memory banks according to their declaration order in the code, the first array starting from the first location in the first bank, the second array starting from the location next to the end of the first array, and so on. In the second mapping, called random mapping, the arrays are assigned to memory locations randomly, making sure that they do not overlap and all elements of a given array are stored consecutively. Experimenting with two initial mappings allows us

Benchmark	Sequential Mapping		Random Mapping	
	8×8MB	4×16MB	8×8MB	4×16MB
adi	92.5	96.0	92.5	96.0
apsi	68.4	61.4	68.2	61.1
btrix	105.0	100.0	105.0	100.0
eflux	92.8	94.3	90.5	91.8
phods	72.7	61.6	70.0	60.3
tomcatv	66.2	60.4	66.0	61.1
vpenta	93.9	83.7	92.8	81.7

Figure 5: Energy consumptions with the mFT policy normalized with respect to the base case.

Benchmark	Migration Threshold		Sampling Threshold	
	MT1	MT2	ST1	ST2
adi	200	400	25	50
apsi	200	400	25	50
btrix	200	400	25	50
eflux	1,000	2,000	100	250
phods	2,000	4,000	250	500
tomcatv	200	400	25	50
vpenta	2,000	4,000	250	500

Figure 6: Migration and sampling thresholds (in units of thousand cycles).

to evaluate the robustness of our data migrator. The base case strategy, on the other hand, statically (at compile time) determines the best array mapping (which is valid throughout the execution of the code); therefore, the initial data mapping does not matter (i.e., it is not taken into account).

In our experiments, we used two memory bank configurations: (i) 8 banks, each 8 MB (denoted 8×8MB), and (ii) 4 banks, each 16 MB (denoted 4×16MB). Note that in both the configurations the total memory capacity is 64 MB. The results in Figure 5 show that array migration reduces the energy consumption by approximately 19%, on the average, over the base case. In only one benchmark, dynamic migration increases the energy consumption (as it increases the number of active banks as compared to the base case). All of these results have been obtained using the sampling and migration thresholds given in Figure 4. We only report the results with the 1FT policy as increasing the number of triplets considered did not make a significant difference for our benchmark codes. More specifically, only in two codes (`eflux` and `vpenta`), the 2FT policy brought a small benefit over the 1FT policy (less than 2.5%). It should also be mentioned that the number of times that the affinity table is probed for migration ranged (in our codes) between 226 (in `apsi`) and 10,468 (in `tomcatv`).

5.2 Sensitivity to the Sampling and Migration Thresholds

To measure the sensitivity of the energy benefits presented above to the sampling and migration thresholds used, we performed another set of experiments (with the 4×16MB configuration and sequential initial layout), where we modified (increased) the sampling and migration thresholds. The new sampling thresholds (ST1 and ST2) and migration thresholds (MT1 and MT2) used are given in Figure 6 for each benchmark in our suite.

The second and third columns in Figure 7 report the (percentage) variation (increase) in energy benefits (with respect to our base sampling thresholds) when the sampling threshold is ST1 and ST2, respectively, while keeping the migration threshold constant at the base value. The fourth and fifth columns give the variation in energy benefits (with respect to our default migration threshold) when we set the migration threshold to MT1 and MT2, respectively (the sampling threshold is kept at its base value). We observe from these results that varying the sampling threshold does not make too much difference whereas migration threshold variations generally generate larger variations. In other words, even larger sampling thresholds allow DM to capture the temporal affinities between arrays. And, working with a large migration thresholds seems to perform

Benchmark	Sampling Threshold		Migration Threshold	
	ST1	ST2	MT1	MT2
adi	-0.5%	-0.6%	0.0%	0.0%
apsi	-1.0%	-1.0%	18.6%	18.8%
btrix	-2.3%	-2.6%	0.0%	0.0%
eflux	0.0%	0.0%	-8.3%	-8.3%
phods	-0.1%	-0.2%	18.5%	19.0%
tomcatv	-2.4%	-2.8%	19.6%	19.6%
vpenta	-0.8%	-0.8%	7.0%	7.0%

Figure 7: Energy sensitivity due to sampling and migration thresholds (4×16MB and sequential initial layout). A positive value (negative value) indicates a reduction (increase) in energy with respect to the original threshold.

'n' value	phods		tomcatv	
	8×8MB	4×16MB	8×8MB	4×16MB
5,000	93.1	88.1	89.4	90.2
10,000	90.7	73.3	76.2	78.6
20,000	82.0	63.2	66.7	**50.0**
40,000	**63.4**	**51.9**	64.0	72.1
80,000	66.1	68.8	**58.3**	83.2
160,000	77.4	76.0	74.7	91.6
320,000	91.2	93.5	89.5	97.4

Figure 8: Energy consumptions with the nLM policy normalized with respect to the base case.

better except for one benchmark (eflux). This is because, in general, a larger migration threshold better distinguishes between false temporal affinities (i.e., affinities with short duration) and real, beneficial temporal affinities.

5.3 Experiments with the nLM Policies

We also conducted experiments with the nLM migration policy. Out of the seven codes we experimented, only two codes (phods and tomcatv) benefited from the nLM policy more than the mFT policy for some 'n' values. The results are given in Figure 8 (the first column gives the 'n' values in 'nLM'). We observe from these results that there exist some 'n' values that generate better results than those obtained through the 1FT policy. For the phods benchmark, this value is 40,000 for both the memory configurations (8×8MB and 4×16MB). For tomcatv, on the other hand, this value is 80,000 for the 8×8MB bank configuration and 20,000 for the 4×16MB bank configuration. Among the values that we experimented with (shown in the first column), these are the optimal ones and are better than the corresponding mFT value. A larger 'n' value delays migration (and sometimes misses the migration opportunity) whereas a smaller 'n' value migrates data too frequently and leads to data freeze (and the associated defrosting cost).

5.4 Energy Overhead of Migration

The cost of energy overhead depends on whether the underlying architecture supports block copy operations that can copy a number of array elements from one portion of the memory to another (in a single instruction). Figure 9 shows the energy overhead (as percentage of the overall memory energy consumption) incurred when different block copy operations are used. Each block copy operation can be defined by the number of array elements that it can copy in a single instruction execution; the larger this number is, the better, as the time (hence, energy) spent during copy is reduced. The x-axis in the figure gives nine alternate block copies (for a given K parameter, the Kth alternative can copy $(64 * K)$ bytes/10 cycles). In the experiments above, we set K to 1. We see from Figure 9 that except for one case, the energy overhead is always below 5%, even if we do not use a very aggressive copy operation. Therefore, for our array-intensive benchmark codes, the runtime energy and

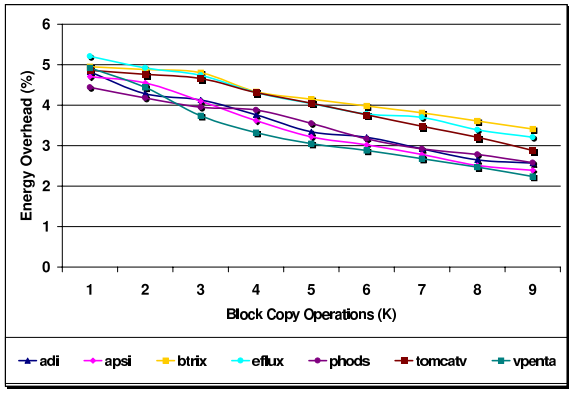

Figure 9: % energy overhead for data migration.

performance benefits due to data migration clearly overwhelms the potential overheads.

6. SUMMARY AND FUTURE WORK

A key design methodology for low-power design is aggressive hardware and software optimizations [2, 1]. Optimizing memory system energy can be crucial in many embedded image and video processing applications. This paper presented an automatic data migration strategy which co-locates arrays with temporal affinity in a small set of memory banks, thereby reducing the energy consumption. Our initial experience with the Data Migrator tool that implements data migration for low power is very promising. The issue of how far the memory system energy can be reduced is interesting to address. Along this direction, we would like to investigate different migration policies and combine these policies with compiler [6] and operating system [7] based techniques to reach a unified solution to the problem.

7. REFERENCES

[1] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2), pp.115-192, April 2000.

[2] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, June, 1998.

[3] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In Proc. *the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[4] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[5] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In Proc. *the International Symposium on Memory Management*, October 1998.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *the 7th International Conference on High Performance Computer Architecture*, Monterrey, Mexico, January 2001.

[7] 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*, November 2000.

[8] J. R. Lorch and A. J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, pp. 60–73, June 1998.

[9] S. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[10] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.