Impact of Data Transformations on Memory Bank Locality

M. Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University, USA
kandemir@cse.psu.edu

Abstract

High-energy consumption presents a problem for sustainable clock frequency and deliverable performance. In particular, memory energy consumption of array-intensive applications can be overwhelming due to poor cache locality. One option for reducing memory energy is to adopt a banked memory architecture, where memory space is divided into banks and each bank can be powered down if it is not in active use. An important issue here is the bank access pattern, which determines opportunities for saving energy. In this paper, we present a compiler-based data layout transformation strategy for increasing the effectiveness of a banked memory architecture. The idea is to transform the array layouts in memory in such a way that two loop iterations executed one after another access the data in the same bank as much as possible; the remaining banks can be placed into a low-power mode. Our simulation-based experiments with nine array-intensive applications show significant savings in memory energy consumption.

1. Introduction and motivation

The PC industry has successfully completed several evolutionary memory transitions: from Fast Paged Mode memory to EDO, to PC66 SDRAM, to PC100 SDRAM. While memory banking has been a widely employed technique in the past for increasing performance, its use in saving energy is relatively new. One important advantage of a banked memory system from the energy consumption viewpoint is that the banks that are not used by the current computation should not be powered up, thereby reducing overall energy consumption.

Our focus is on a banked memory architecture where each bank can be power-controlled independently. More specifically, each bank can be placed into low-power operating mode when it is not used by the current computation. In an RDRAM-like architecture, one may have multiple low-power modes (states) to choose from when a memory bank is detected to be idle. A major tradeoff between these different modes is between the energy saving (while in the low-power mode) and resynchronization cost (i.e., the time it takes to bring back a memory bank from the low-power state to the fully-operational active state).

Figure 1 shows typical low-power operating modes and transitions between them. The values associated with the nodes correspond to the per cycle energy consumption for the bank, whereas the values associated with the edges indicate the resynchronization costs. These values clearly illustrate the tradeoff between energy and performance. Specifically, a more energy saving operating mode also incurs a higher resynchronization cost.

It should be observed that the benefits from low-power operating modes can increase if, somehow, idleness of the bank could be increased. This is because in this case the resynchronization cost can be compensated by significant savings in energy. In comparison, a short idleness either will not allow us to place the bank into the most energy-saving mode (i.e., the one that consumes the smallest amount of energy per cycle), or will incur large performance penalty. Consequently, an important goal is to increase idleness of memory banks as much as possible. This can be achieved by smart data allocations to memory banks and/or re-ordering computations. In this study, we make a case for compiler-oriented data layout transformations for array data since it can increase the effectiveness of low-power modes available in the memory system. We also need to mention that in this work we assume that no virtual memory support exists in the system under consideration. Consequently, the compiler can directly work with physical addresses; that is, it can layout data in physical memory and place banks into low-power modes based on the information it collects during program analysis. Note that there exist many embedded systems that work without a virtual memory support [5]. Work is in progress to extend the techniques discussed in this paper to environments with virtual memory (by enlisting help from the operating system (OS)).

Previous research shows that compiler-based (e.g., [2]), OS-based (e.g., [8]), and pure hardware-based schemes (e.g., [2]) are possible to decide the most suitable low-power mode to use when a memory bank is detected to be idle. Since in this work we focus on array-intensive applications, we opted to use a compiler-based approach, where an optimizing compiler (taking into account loop access patterns and array-to-bank mappings – that is, the layout of data in banked memory) decides which operating mode to use. Note that (where applicable) such a compiler-based strategy has an important advantage over pure OS-based and hardware-based techniques. Specifically, the compiler-based strategy (unlike pure OS or pure hardware-based strategies) does not rely on history of data access patterns; that is, the compiler can predict (quite accurately for the array-intensive codes) future data access patterns (and also, future idle times), and select the most appropriate mode to switch to when idleness is predicted. In addition, the compiler can also predict when an idle bank will be requested in the future, and can pre-activate it in an attempt to eliminate re-synchronization latency. Details of the compiler-based low-power mode detection strategy are beyond the scope of this paper. It should be noted, however, that the compiler-based

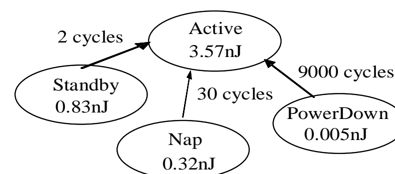

Fig 1. Operating modes and transitions between them.

1530-1591/04 $20.00 (c) 2004 IEEE

strategy selects the most suitable operating modes for a given access pattern and memory layout. In this work, *we modify data layouts in memory to allow the compiler place more banks into low-power mode and/or keep banks in low-power operating modes longer*. In other words, *our approach is geared towards increasing the effectiveness of compiler-directed low-power mode management*. While previous research employed data layout transformations for cache locality [11, 9], in this paper, we use them for energy optimization in banked memory architectures.

[3] studies OS-based DRAM power control policies. [7] evaluates the impact of classical loop optimizations on energy consumption of banked memories. [6] presents an iteration space reordering technique for banked memories. In contrast, the work presented in this paper is oriented toward increasing the benefits of low-power modes by data distributions across memory banks. [4] shows how a sleep mode can be exploited for memory partitions. [15] and [14] discuss techniques for exploiting dual banks for ASIPs and DSPs, respectively. [12] addresses the topic of incorporating the application-specific customization of memory bank configuration into behavioral synthesis. In comparison, we study how compiler-directed data optimization can improve energy behavior of a multi-banked system.

The rest of this paper is organized as follows. Section 2 presents background on iteration space and data space representations for array-intensive applications. Section 3 explains array-to-virtual bank mappings and Section 4 discusses virtual bank-to-physical bank mapping. Section 5 shows how data transformations can be useful in increasing the effectiveness of low-power operating modes. Section 6 introduces our experimental platform, and presents data that show the effectiveness of our strategy. Section 7 gives our concluding remarks.

2. Preliminaries

We can define iterations of a loop nest as a set, each element of which corresponds to an iteration vector. Each execution of loop body uses a vector from this set. Given this, an array access within a loop nest can be expressed as RI + r, where R is the access matrix, I is the iteration vector, and r is an offset vector [16, 10]. As an example, for an array reference such as A(i-1,j+2) that occurs within a loop nest where i is the outer loop and j is the inner, R is the identity matrix, I = (i j)T, and r = (-1 2)T. It is to be noted that each iteration vector I accesses an element via this array reference. An array element accessed by an iteration vector represents an index vector. In the example above, this vector is a = (i-1 j+1)T. It can be observed that if the nest in question has n loops and the reference in question belongs to a k-dimensional array, R is a k-by-n matrix, I is an n-entry vector, and r is a k-entry vector.

Informally, a data transformation indicates a mapping of the index space. In mathematical terms, a data transformation can be represented using a pair (M,m), and it transforms an original index vector RI+r to MRI+Mr + m. If we restrict ourselves to dimension-preserving data transformations, M is a k-by-k matrix and m is a k-entry vector (for a k-dimensional array). For instance, assuming that

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad m = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

the index vector $a = (i\text{-}1 \quad j\text{+}1)^T$ is mapped to $(j\text{+}1 \quad i\text{-}1)^T$.

3. Array-to-virtual bank mapping

In our framework, array elements are mapped to (physical) memory banks using a two-level mapping. In the first level, an index vector (of an array) is mapped to a virtual bank, and in the second level, each virtual bank is mapped to a physical bank. This two-level process is depicted in Figure 2.

The compiler operates under the assumption of a virtual bank space (VBS), which can be multi-dimensional. Given an array index vector a, we find the virtual bank it is mapped using an affine mapping $\theta a + \varphi$. Therefore, two different elements (of the same array) represented by index vectors a and b map onto the same virtual bank if and only if:

$$\theta a + \varphi = \theta b + \varphi \Rightarrow \theta a = \theta b \Rightarrow \theta(a-b) = 0.$$

In other words, (a-b) should be in the kernel set of θ. In this paper, the pair (θ,φ) is called the bank mapping. Note that different criteria can be used in determining suitable bank mappings, and each array can have a different bank mapping than the other arrays in a given application.

4. Virtual bank-to-physical bank mapping

A virtual bank-to-physical bank mapping (or a physical mapping for short) determines how virtual banks are mapped to the physical banks in the architecture. Let v be a virtual bank. The corresponding physical bank can be determined using a mapping such as: $\xi v + \sigma$. Now, in order for two different array elements a and b to be mapped to the same physical bank, one should have:

$$\xi\theta a + \xi\varphi + \sigma = \xi\theta b + \xi\varphi + \sigma \Rightarrow \xi\theta a = \xi\theta b \Rightarrow \xi\theta\,(a-b) = 0.$$

There is a good reason to adopt such a two-level mapping (instead of using a more intuitive one-level mapping that maps arrays directly to the physical banks). In many cases, we want a compiler optimization to be easily portable to another platform without much difficulty. Because of this, it makes sense to work with VBS rather than PBS (physical bank space). In other words, we can write our compiler-based optimization strategy only once (using the VBS as the reference point), and when we want to port it to other architectures (with different physical bank structures), we only need to change the virtual bank-to-physical bank mapping. Note that, in general, the virtual bank-to-physical bank mapping can reduce the dimensionality and/or extents (dimension sizes) of the VBS. Informally, a mapping is specified by giving a decomposition style for each dimension of the virtual bank space along with the physical bank size in each dimension (of the physical bank space). For example, a mapping such as

$$(b_1,b_2,b_3) \rightarrow (b_1/p,*,*)$$

indicates that a three-dimensional virtual bank space is mapped to a one-dimensional physical bank space. The * notation indicates that the corresponding virtual bank dimension is not

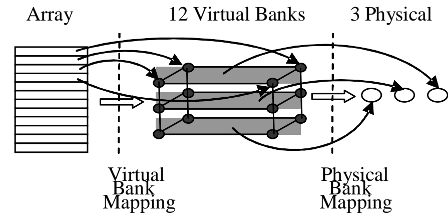

Fig 2. Two-level mapping of an array into memory banks.

distributed across the physical banks. This means, for the mapping above, that the second and third dimensions are not distributed; instead, they are folded. The notation b_1/p (where / denotes integer division) in the first dimension, on the other hand, reveals that this dimension (of the virtual bank space) is distributed across p physical banks. So, assuming p is 8, under such a mapping, the virtual bank (16,i,j) is mapped to physical bank 2 for all values i and j can take. In terms of matrices, such a mapping can be expressed as: $\xi = (1/p\ \ 0\ \ 0)$ and $\sigma = 0$.

It should be noted that this mapping function definition is quite general and encompasses very different types of virtual bank-to-physical bank mappings. For example, it can accommodate functions such as

$$(b_1,b_2,b_3) \rightarrow (b_1/p,b_2/q,*)$$

which indicates that a three-dimensional virtual bank space is mapped to a two-dimensional physical bank space that contains pxq banks (this might be useful, for example, for some SDRAMs where memory banks form actually a two-dimensional grid). It should also be noted that when multiple virtual banks are mapped to the same physical bank, the loop iterations that access those virtual banks are localized (that is, they exhibit bank locality as they now – after folding – access the same physical bank). Therefore, selection of a suitable mapping function can be important. A virtual bank-to-physical bank mapping tries to take advantage of spatial locality between neighboring banks. However, from the compiler's perspective, it should be sufficient to work with the VBS instead of the PBS. This is because whenever we achieve access locality for a virtual bank, it is guaranteed that that locality will extend to the PBS as well since a virtual bank is mapped to only a single physical bank. Therefore, optimizing bank locality in the VBS is sufficient for our purposes. The rest of this paper discusses our approach to optimizing bank locality in the VBS.

5. Role of data transformations in bank locality

As discussed earlier, our focus in this paper is on studying the cases where a data transformation might be of use in exploiting bank locality for array-intensive applications. Let us start by defining formally what we mean by bank locality.

Definition: If two iteration vectors, say I and J, are very close to each other (that is, I-J = a lexicographically small value, preferably $(0\ \ 0\ \ 0\ \dots\ 0\ \ 1)^T$), they are said to have *temporal affinity*.

Definition: If two iteration vectors with temporal affinity access the array elements in the same virtual bank, they are said to exhibit *bank locality*.

Now, let us determine the condition for bank locality.

Let $I^+ = I + (0\ \ 0\ \ 0\ \dots\ 0\ \ 1)^T$, where + denotes vector addition. In order to have bank locality, the array elements accessed by I and I^+ (via the same array reference in the code) should be on the same virtual bank. In mathematical terms, we need to have:

$$\theta(RI + r) + \varphi = \theta(RI^+ + r) + \varphi \Rightarrow \theta RI = \theta RI^+ \Rightarrow\ \theta h = 0,$$

where h is the last column of R. This type of bank locality can be termed as *intra-reference bank locality*, i.e., the bank locality that originates from a single array reference in the application code.

It is important to note here that for a given θ matrix, the vector h may or may not be in its kernel set. Therefore, it is not guaranteed that we can achieve intra-reference bank locality. Now, let us assume that we use a data transformation represented by (M,m) in the array in question. In this case, rewriting the condition for intra-reference bank locality, one might have:

$$\theta(MRI + Mr + m) + \varphi = \theta(MRI^+ + Mr + m) + \varphi$$
$$\Rightarrow \theta MRI = \theta MRI^+ \Rightarrow \theta Mh = 0.$$

It is to be observed that, now, we have a flexibility of selecting a suitable M such that h is in the kernel set of θM. Therefore, we can conclude that data transformations can be useful for achieving intra-reference bank locality.

Example: Let us assume an array reference A(i+j+1,j-1) within a loop nest with two loops: i (outer) and j (inner). It is easy to see that:

$$R = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad r = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Assuming that $\theta = (1\ \ 0)$, we can see that

$$\theta h\ =\ (1\ \ 0)\begin{pmatrix} 1 \\ 1 \end{pmatrix}\ = 1\ \neq 0.$$

Therefore, we can conclude that it is not possible to exploit intra-reference bank locality under this distribution (bank mapping). However, if we use a data transformation matrix M, from

$$\theta Mh = (1\ \ 0)\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad M= (1\ \ 0)\begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 0,$$

we can find that m11 + m12 = 0. A solution to this last equation is m11 = 1 and m12 = -1, which can subsequently be completed to a full data transformation matrix

$$M = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix}.$$

In other words, it is possible to find an M matrix to satisfy intra-reference bank locality. This small example illustrates how useful a data transformation can be in optimizing bank locality.

We next focus on *inter-reference bank locality*. Let RI + r and R'I + r' be two different references to the same array. In order to have inter-reference bank locality, we should have:

$$\theta(RI + r) + \varphi = \theta(R'I + r') + \varphi \Rightarrow \theta(R-R')I = \theta(r'-r).$$

Let us consider two cases:

Case I. R = R'. This represents a very common case in array-intensive embedded image/video applications. In this case, the relation above reduces to θ(r'-r) = 0. Consequently, if r'-r is not in the kernel set of θ, we cannot have inter-reference bank locality. On the other hand, if we employ a data transformation represented by (M,m), we have

$$\theta(MRI + Mr + m) + \varphi = \theta(MR'I + Mr' + m) + \varphi$$
$$\Rightarrow \theta M(R-R')I = \theta M(r'-r).$$

Since, we have R = R', this last equation reduces to θM(r'-r)=0. Now, it may be possible to select a suitable M such that r'-r is in the kernel set of θM. That is, data transformation increases the chances for inter-reference bank locality. To illustrate how this works in practice, we consider the following example.

Example: Let us assume two array references, A(i+j+1,j-1) and A(i, j), within a loop nest with two loops: i (outer) and j (inner). Assuming, as before, that we use $\theta = (1\ \ 0)$ as our bank mapping, we can find that

$$\theta(r'-r) = (1\ \ 0)\ \{\begin{pmatrix} 1 \\ -1 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix}\} = (1\ \ 0)\begin{pmatrix} 1 \\ -1 \end{pmatrix}\ = 1.$$

Since θ(r'-r) ≠ 0, it is not possible to satisfy inter-reference bank locality. On the other hand, if we are allowed to use a data transformation matrix M, from

$$\theta M(r'-r) = (1\ \ 0)\begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = 0,$$

we have m11 - m12 = 0. A possible solution is m11=1 and m12 =1, which can subsequently be completed to a full data transformation matrix

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

So, it is possible to obtain inter-reference bank locality using this data transformation. This example clearly illustrates that data transformations can be very useful in exploiting inter-reference bank locality.

Case II. $R \neq R'$. In this case, if we do not use any data transformation, we have $\theta(R-R')I = \theta(r'-r)$, as determined above. So, there is no way that this equality can be satisfied since the right side is constant while the left side can take different values for different iteration vectors I. However, if we use a data transformation (M,m), we need to satisfy $\theta M(R-R')I = \theta M(r'-r)$. This can be achieved by satisfying the following two equalities:

- $\theta M(R-R') = 0$, and
- $\theta M(r'-r) = 0$.

That is, even in this case, it might be possible to find an M matrix to satisfy these two constraints at the same time, and thus obtain inter-reference bank locality.

So far, we have only considered bank locality problem from the perspective of a single array (that is, *intra-array bank locality* whether it is intra-reference or inter-reference). It is also possible to exploit *inter-array bank locality*. Let as assume that RI + r and R'I + r' are references to two different arrays. For inter-array bank locality, one should have:

$$\theta_1(RI + r) + \varphi_1 = \theta_2(R'I + r') + \varphi_2$$
$$\Rightarrow (\theta_1 R-\theta_2 R')I = \theta_2 r' - \theta_1 r + \varphi_2 - \varphi_1.$$

In this formulation, (θ_1, φ_1) and (θ_2, φ_2) represent array-to-virtual bank mappings for the two arrays under consideration. If $\theta_1 R-\theta_2 R' = 0$, then the above equation gets reduced to:

$$\theta_2 r' - \theta_1 r + \varphi_2 - \varphi_1 = 0.$$

However, in the general case $\theta_1 R-\theta_2 R' \neq 0$, and as a result, it is not possible to satisfy this equation.

On the other hand, if we assume data transformations (M_1,m_1) and (M_2,m_2) for these two arrays, we need to satisfy:

$$\theta_1(M_1 RI + M_1 r + m_1) + \varphi_1 = \theta_2(M_2 R'I + M_2 r' + m_2) + \varphi_2$$
$$\Rightarrow (\theta_1 M_1 R-\theta_2 M_2 R')I = \theta_2 M_2 r' - \theta_1 M_1 r + \theta_2 m_2 - \theta_1 m_1 + \varphi_2 - \varphi_1.$$

This last equality can be satisfied if one can satisfy the following two equations:

- $\theta_1 M_1 R-\theta_2 M_2 R' = 0$, and
- $\theta_2 M_2 r' - \theta_1 M_1 r + \theta_2 m_2 - \theta_1 m_1 + \varphi_2 - \varphi_1$

In fact, if, using the first equality, we can find M_1 and M_2 matrices, and we can substitute them in the second equality and solve it for m_1 and m_2.

6. Experiments

6.1 Setup

All energy numbers presented in this paper have been obtained using a custom memory energy simulator. This simulator takes as input a C program and a banked memory description (i.e., the number and sizes of memory banks as well as available low-power operating modes with their energy saving factors and re-synchronization costs). As output, it gives the energy consumption in memory banks along with a detailed bank inter-access time profiles. By giving original and optimized programs to this simulator as input, we measure the impact of our data transformation strategy on memory system energy.

The data transformation framework presented in this paper has been fully implemented using the SUIF infrastructure from Stanford University [1]. SUIF has independently developed compilation passes that work together by using a common intermediate format (IF) to represent programs. A typical compilation framework based on SUIF includes the following components: front end, data dependence analysis, and several optimization modules. Our framework is implemented as a separate optimization module within SUIF. We also use a powerful back-end compiler (when converting the C code to executable) that performs instruction scheduling and graph coloring-based global register allocation. Unless stated otherwise, 8x8MB (that is, 8 memory banks, each has a capacity of 8MB) is our default bank configuration. Note that if a bank is not accessed during the execution of an application, it is never activated for both the original and the optimized code versions. In other words, even our base case takes advantage of low-power operating modes, and the approach proposed in this paper tries to improve over it. That is, *all the energy benefits reported in this work are coming from our data layout optimization strategy*. Also, we use a default array-to-bank mapping in most of our experiments. In this default mapping, each array is laid out in memory in a row-major fashion, and the next array (in the program declaration part) is stored starting from the location next to the one where the previous array ends (that is, the arrays are stored in memory one after another). However, we also report some results with smarter array mappings (distributions). While in this work we use the energy consumption and resynchronization values shown in Figure 1, our framework is general enough in that it can work with different set of low-power modes as well. The energy values shown in Figure 1 have been obtained from the measured current values associated with memory banks documented in memory data sheets (for 3.3V, 2.5 nsec cycle time, 8MB memory) [13]. The re-synchronization latencies have also been obtained from the same data sheets. Based on the trends gleaned from the data sheets, the energy values are increased by 30% when bank size is doubled. Unless stated otherwise, our architecture does not have a data cache (since we want to isolate the energy benefits in the banked memory system). However, later in this section we also report experimental data with different data cache sizes. In fact, our results indicate that the proposed strategy is successful with both cacheless and cache-based systems.

Table 1. Benchmark codes used in this study.

Benchmark Name	Dataset Size (KB)	Execution Cycles (M)	Energy Consumption (mJ)
oreg	728.60	482.22	244.08
adi	555.94	396.07	180.84
full-search	624.00	474.33	209.38
hier	624.00	412.69	191.47
mxm	1,280.00	681.53	424.49
compress	876.14	619.90	320.99
tomcatv	767.80	560.16	274.37
jacobi	1,018.00	677.77	387.65
red-black SOR	1,018.00	790.05	479.44

6.2 Results

To evaluate our strategy quantitatively, we performed experiments with nine array-intensive benchmarks. Important characteristics of these codes are listed in Table 1. The third column gives the execution cycles, while the fourth column shows the memory energy consumption without our optimization.

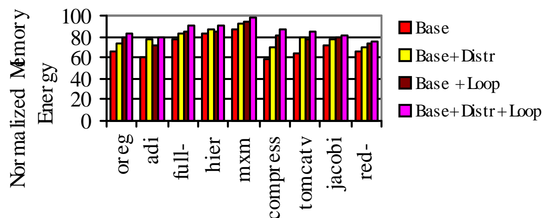

Fig 3. Normalized energy consumption with respect to different versions.

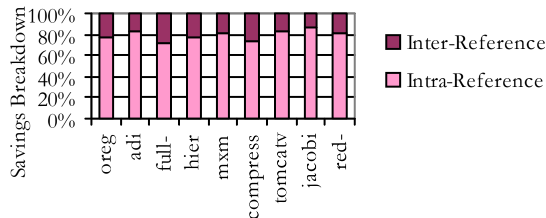

Fig 4. Breakdown of energy savings.

As mentioned earlier, even this baseline version makes full use of low-power operating modes available in the architecture.

The first bar for each benchmark in Figure 3 (called Base) gives the energy consumption due to our strategy, normalized with respect to the default array distribution without any program transformation (the last column in Table 1). We see that our data optimization brings 29.6% improvement on the average. We also note that relative energy savings depends on the benchmark used. For example, the savings with some benchmarks such as hier are not as good as those with the others, mainly due to fact that the compiler was not able to select good data transformations for the arrays in these codes. The main reason for this is that the references to the same array create conflicts that prevent the compiler from using the ideal data transformation matrix (M) for the array in question. Still, even with such benchmarks, our approach achieves energy savings around 15%. In comparison, in benchmarks such as compress, there are few references per array; hence fewer chances for conflict in selecting the most appropriate data transformation.

While our energy savings are significant, one might argue that data distribution (across the memory banks) also plays a key role in shaping energy consumption behavior. So, we also measured the energy savings with respect to a distribution-optimized code. The specific data distribution algorithm (that is, the algorithm that decides which arrays should be mapped to which banks) is from [2]. The second bar for each benchmark in the figure (named Base+Distr) shows the normalized energy consumption of our strategy. The average energy reduction is around 21.2%, indicating that our approach is still very successful in optimizing memory energy behavior. The third bar for each code (named Base+Loop) gives the normalized energy consumption of our strategy with respect to a version that uses the default array distribution and data locality oriented loop transformation. The rationale behind this version is that locality oriented loop transformations in general improve spatial access patterns, and this can also improve bank locality. The specific loop transformation strategy that is used here is from [10]. We see that the average energy savings brought by our approach with respect to this version is around 19.1%. What this result says is that the data transformation strategy complements the loop transformation based optimization. Finally, the last bar in the figure (referred to as Base+Distr+Loop) shows the normalized energy consumption of our scheme with respect to a strategy that uses both optimized data distribution and loop transformation. Even against this highly-optimized version we achieve 14.7% energy savings on the average when considering all codes in our benchmark suite. Overall, these results clearly show that our data transformation based approach is very effective in increasing the effectiveness of low-power operating modes.

It should be emphasized that the energy benefits shown in Figure 3 have been obtained by trying to satisfy both intra-reference constraints and inter-reference constraints. To illustrate individual contributions coming from these different types of localities, we show in Figure 4 how the energy benefits are broken down (the results are normalized with respect to the Base+Distr+Loop version). One can observe from the trends shown in this graph that most of the energy savings are coming from optimizing intra-reference bank locality. The main reason for this behavior is that satisfying intra-reference locality brings more benefits since this captures access behavior across loop iterations. This is in contrast to inter-reference locality whose impact is limited by the number of references to the same array in the loop body. Nevertheless, we still observe that the contribution of satisfying inter-reference constraints to overall memory energy savings is around 20.8% on the average, which indicates that it is important to take care of them as well.

An important parameter that influences the magnitude of energy savings is the number of memory banks. This is because a larger number of banks give a finer-granular control to the compiler to place memory regions into low-power operating modes. In order to quantify the impact of our approach with different bank configurations, we performed another set of experiments. More specifically, keeping the total memory size fixed at 64MB, we conducted experiments with 2, 4, 8, 16, and 32 banks. The results are given in Figure 5 (again as values normalized with respect to the Base+Distr+Loop version). We can observe from these results that working with larger number of banks (i.e., with smaller bank sizes) in general increases the energy benefits coming from the proposed data layout optimization strategy. This is because, as indicated above, smaller bank sizes give our strategy more opportunities for energy-managing even smaller portions of main memory. Such a finer-grain management, in turn, increases the energy benefits. However, we also note that in some applications, increasing the number of banks beyond a specific number does not increase savings. This occurs because of the data access pattern of such applications. Specifically, the access pattern of those applications

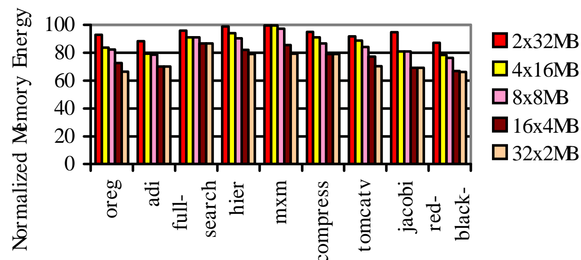

Fig 5. Normalized memory energy consumption with varying number of banks.

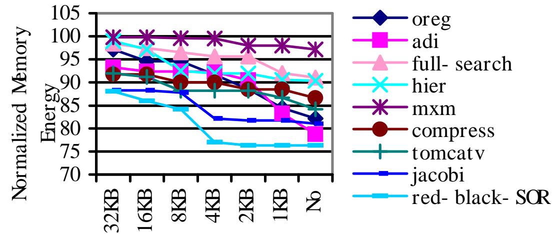

Fig 6. Normalized memory energy consumption with varying data cache sizes.

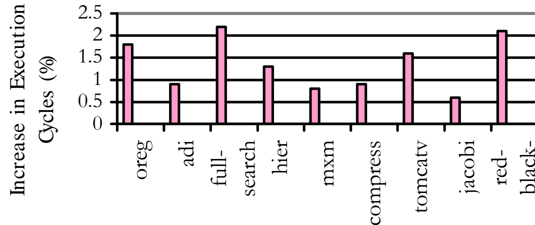

Fig 7. Percentage increase in execution cycles due to our data transformation strategy.

spans more banks when the number of banks is increased beyond a specific value.

In our experiments so far we have focused on a memory architecture without data cache. Including a cache in the hierarchy can filter some requests, thereby increasing the idleness of memory banks. However, since even unoptimized codes benefit such filtering, we can expect some reduction in energy savings. The result shown in Figure 6 corroborates this expectation. Still, even with a 16KB data cache, we obtain an average energy saving of roughly 7% over the Base+Distr+Loop version (which itself is highly optimized). Thus, our data optimization is beneficial even with data caches. It should also be emphasized that the Base+Distr+Loop version is already a highly optimized version, and it is really difficult to further improve its energy behavior.

While the experimental data presented so far clearly demonstrate the energy benefits of our strategy, to be fair, one needs to consider performance impact as well. Therefore, in Figure 7, we give the percentage increase in original execution cycles (i.e., the cycles when no power control is present) when the proposed data transformation is used. Overall, one can see that the increase in execution cycles varies from 0.79% to 2.21% depending on the benchmark used, averaging in 1.35%. The reason that we do not incur much performance penalty is that the compiler pre-activates a memory bank before it is actually needed. Note that this is possible in our application domain since (considering the array-to-bank mappings) the compiler can accurately predict the next access to a given bank. This bank pre-activation strategy in turn limits the potential degradation in performance.

7. Concluding remarks

Energy consumption is becoming a first-order design parameter as processor-based systems continue to become more and more complex. Off-chip memory energy consumption in particular can be a limiting factor in many system designs. In this work, we focus on executing array-intensive benchmarks in banked memory architectures, and propose a compiler-directed strategy that modifies data layouts in memory to place more memory banks into low-power mode and/or keep memory banks in low-power operating modes longer. The experimental data obtained using nine array-intensive benchmarks and a simulation environment show the potential of our approach in saving memory energy.

8. References

[1] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a Software Perspective. IEEE Micro, June 1996, pages 52-61.

[2] 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 Int'l Symposium on High Performance Computer Architecture, 2001.

[3] X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of DRAM Power Control Policies Using Deterministic and Stochastic Petri Nets, In Proc. Workshop on Power-Aware Computer Systems, Springer-Verlag, February, 2002.

[4] A. Farrahi, G. Tellez, and M. Sarrafzadeh. Exploiting Sleep Mode for Memory Partitions and Other Applications, VLSI Design, Vol. 7, No. 3, pp. 271-287.

[5] W-M. W. Hwu. Embedded microprocessor comparison. http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[6] M. Kandemir, U. Sezer, and V. Delaluz. Improving Memory Energy Using Access Pattern Classification. In Proc. the International Conference on Computer Aided Design, San Jose, CA, November 4-8, 2001.

[7] M. Kandemir, I. Kolcu, and I. Kadayif. Influence of Loop Optimizations on Energy Consumption of Multi-Bank Memory Systems. In Proc. International Conference on Compiler Construction, Grenoble, France, April 6-14, 2002.

[8] 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.

[9] S. -T. Leung and J. Zahorjan. Optimizing Data Locality by Array Restructuring. Technical Report TR 95-09-01, Dept. of Computer Science and Engineering, University of Washington, September 1995.

[10] W. Li. Compiling for NUMA Parallel Machines. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993.

[11] M. O'Boyle and P. Knijnenburg. Integrating Loop and Data Transformations for Global Optimization. In Proc. International Conference on Parallel Architectures and Compilation Techniques, October 1998, Paris, France.

[12] P. R. Panda. Memory Bank Customization and Assignment in Behavioral Synthesis. In Proc. ICCAD, 1999.

[13] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[14] M. A. R. Saghir, P. Chow and C. G. Lee, Exploiting dual data-memory banks in digital signal processors. In Proc. International conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, pp. 234–243.

[15] A. Sudarsanam, S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. ACM Transactions on Design Automation of Electronic Systems 5, 2000, pp. 242–264.

[16] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In Proc. ACM Conference on Programming Language Design and Implementation, pp. 30-44, June 1991.

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