Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memories[*]

V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu[†]
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802, USA
mdl@cse.psu.edu

Abstract

With the increased use of embedded/portable devices such as smart cellular phones, pagers, PDAs, hand-held computers, and CD players, improving energy efficiency is becoming a critical issue. To develop a truly energy-efficient system, energy constraints should be taken into account early in the design process; i.e., at the source level in software compilation and behavioral level in hardware compilation. Source-level optimizations are particularly important in data-dominated media applications that have become pervasive in energy-constrained mobile environments.

This paper focuses on improving the effectiveness of energy savings from using multiple low-power operating modes provided in current memory modules. We propose a source-level data space transformation technique called array interleaving that colocates simultaneously used array elements in a small set of memory modules. We validate the effectiveness of this transformation using a set of array-dominated benchmarks and observe significant savings in memory energy.

1 Introduction

Data-dominated media applications have become pervasive in energy-constrained mobile environments. Systems running such applications have been found to consume a significant portion of their energy budget in the memory hierarchy [1, 11]. Significant strides made in the low-power memory design encompassing circuit and architectural techniques have helped to partially alleviate this problem. One such technique is the provision of multiple low-power operating modes through partial shutdown of the memory modules when they are not in active use. Many current memory systems (e.g., [9]) provide mechanisms for utilizing such power modes to save energy.

The effectiveness of power mode control schemes depends critically on the memory access patterns and data allocation strategies in these memories. In particular, a poor data allocation strategy can lead to large energy loss by keeping large number of modules active most of the time. An optimization strategy should try to cluster data with temporal affinity in a small set of memory modules and turn off the remaining modules to save energy. In this paper, we focus on improving the effectiveness of these low-power operating modes through a data space (array layout) optimization framework. More specifically, we propose an array interleaving mechanism that clusters the data elements of multiple arrays accessed simultaneously into a single common data space so that fewer memory modules need to be active at a given time. This mechanism is particularly useful in embedded signal and video processing environments that use array structures extensively. Given an application, the proposed mechanism automatically determines the arrays to be interleaved, and also transforms the code accordingly by replacing the original array references and declarations with their transformed equivalents.

The proposed mechanism has been evaluated using a set of array-based applications. We have utilized a cycle-accurate simulator developed in-house to model the energy and performance behavior of a memory architecture with low-power operating modes. The simulator models three different mode control mechanisms with varying degrees of sophistication. Our preliminary results indicate significant savings in memory energy. Based on these results, we conclude that array interleaving is very beneficial from an energy viewpoint and should be supported by compilers targeting multi-bank memory systems.

The remainder of the paper is organized as follows. The next section discusses the memory system architecture we considered and the low-power operating modes we used.

[*] This work was supported in part by Grants from GSRC and NSF CAREER Awards 0093082 and 0093085

[†] Computation Department, UMIST, Manchester M60 1QD, UK.

	Energy Consumption (nJ)	Re-synchronization Time (cycles)
Active	3.570	0
Standby	0.830	2
Napping	0.320	30
Power-Down	0.005	9,000
Disabled	0.000	NA

Figure 1. Energy consumption and re-synchronization times for different operating modes.

Section 3 gives a description of array interleaving. Our energy optimization strategy (based on array interleaving) is explained in Section 4. The experimental results are presented in Section 5. Finally, we summarize the contributions of this work in Section 6.

2 Memory Architecture

The target system for our approach is a memory system that contains a number of modules organized into banks (rows) and columns. We refer to such banked architectures as partitioned-memory (or banked-memory) architectures. Accessing a word of data would require activating the corresponding bank and columns of the shown architecture. There are several ways of saving energy in such a memory organization. The approach adopted in this paper is to put the unused memory banks into a low-power operating mode.

In all our experiments, we use one module in a bank and hence, use the terms bank and module interchangeably. Note that we are assuming a RAMBUS style of memory [8] which obviates the need for conventional interleaving. Each bank operates independently, and when not in active use, it can be placed into a low-power operating mode to conserve energy. This paper considers only dynamic energy consumption and does not account for leakage current. In addition to the optimization proposed in this paper, it is also possible to apply leakage energy reduction techniques to unused memory modules. Each operating mode works by activating specific portions of the memory circuitry such as column decoders, row decoders, clock synchronization circuitry and refresh circuitry [8], and can be described using two related metrics: *energy consumption* and *re-synchronization time*. The energy consumption is the amount of energy consumed per cycle in a given operating mode. The re-synchronization time is the time (in cycles) it takes to bring a bank from a low-power mode to the active (fully-operational) mode. Typically, lesser the energy consumption, higher the re-synchronization time. Consequently, the selection of low-power operating mode has both energy and performance impacts and usually involves a tradeoff between them.

For the purposes of this paper, we assume five different operating modes: an active mode (the only mode during which the memory read or write activity can occur) and four low-power modes, namely, standby, napping, power-down, and disabled. Current DRAMs [9] support up to six energy modes of operation with a few of them supporting only two modes. We collapse the read, write, and active without read or write modes into a single mode (called active mode) in our experimentation. However, one may choose to vary the number of modes based on the target DRAM architecture and specification. The energy consumptions and re-synchronization times (to bring the module back to active mode) for these operating modes are given in Figure 1. The energy values shown in this figure have been obtained from the measured current values associated with memory modules documented in memory data sheets (for a 3.3V, 2.5ns cycle time, 8MB memory) [8]. The re-synchronization times are also obtained from data sheets. Based on trends gleaned from data sheets, the energy values are increased by 30% when module size is doubled.

Typically, several of the DRAM modules are controlled by a memory controller which interfaces with the memory bus. The interface is used not only for latching the data and addresses, but is also used to control the configuration and operation of the individual memory modules as well as their operating modes. The controllers contain some prediction hardware to estimate the time until the next access to a memory module and circuitry to ask the memory controller to initiate mode transitions. A limited amount of such self-monitored power-down is already present in current memory controllers (e.g., Intel 82443BX and Intel 820 Chip Set). In this paper, we utilize three such predictors: constant threshold predictor (CTP), adaptive threshold predictor (ATP), and history-based predictor (HBP).

The CTP mechanism is similar to the mechanisms used in current memory controllers. After 10 cycles of idleness, the corresponding module is put in standby mode. Subsequently, if the module is not referenced for another 100 cycles, it is transitioned into the napping mode. Finally, if the module is not referenced for a further 1,000,000 cycles, it is put into power-down mode. We do not utilize the disabled state that shuts off the refresh circuitry to avoid loss of data in the memory modules. Whenever the module is referenced, it is brought back into the active mode incurring the corresponding re-synchronization costs (based on what mode it was in). The other two schemes are enhancements to the CTP mechanism. The ATP scheme dynamically adapts the thresholds to adjust for any mispredictions it may have made. The HBP scheme maintains a history of the operating mode changes to predict the future mode transitions more accurately. These schemes are adapted from [2], and their details are beyond the scope of this paper. In addition, to keep the issue tractable, this paper bases the experimental results on a single program environment and does not consider the virtual memory system (i.e., we assume that the compiler directly deals with physical addresses). Note that many embedded environments [4] operate without any virtual memory support. The mode control capabilities in the DRAM have also been explored recently for developing novel power-aware page allocation policies [6].

3 Array Interleaving

Array interleaving is a data space (array layout) transformation technique that takes multiple arrays, and maps them into a single array. This mapping should be one-to-one (i.e., each array element should be mapped into a unique place, and no two array elements should be mapped into the same place in the new array) and, after the mapping, the array references in the program and array declarations should be modified (re-written) accordingly. Consider the following loop nest that accesses two one-dimensional arrays using the same subscript function.

Example 1:

```
for ($i$=1;$i$ $\leq$N;$i$++)
  $b+ = U_1[i] + U_2[i]$;
```

We consider the following mapping of arrays U_1 and U_2 to the common data space (array) X.

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_2[i] \longrightarrow X[2i],$$

in which case we can re-write the loop nest as follows:

```
for ($i$=1;$i$ $\leq$N;$i$++)
  $b+ = X[2i-1] + X[2i]$;
```

A pictorial representation of this mapping is given in Figure 2(i). Note that, in this new (transformed) nest above, for a given loop iteration, two references access two consecutive array elements whereas the original code accesses two nonconsecutive elements (each from a different array) for a given iteration execution. The same transformation can be applied to multidimensional arrays as well. As an example, consider the following two-level nested loop:

Example 2:

```
for ($i$=1;$i$ $\leq$N;$i$++)
  for ($j$=1;$j$ $\leq$N;$j$++)
    $c+ = U_1[i][j] + U_2[i][j]$;
```

If we use the data transformations

$$U_1[i][j] \longrightarrow X[i][2j-1] \text{ and } U_2[i][j] \longrightarrow X[i][2j],$$

we obtain the following nest:

```
for ($i$=1;$i$ $\leq$N;$i$++)
  for ($j$=1;$j$ $\leq$N;$j$++)
    $c+ = X[i][2j-1] + X[i][2j]$;
```

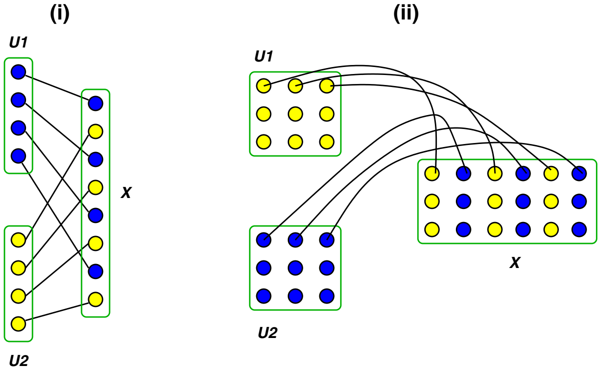

Figure 2. (i) Interleaving two one-dimensional arrays; (ii) Interleaving two two-dimensional arrays.

This transformation can be viewed as converting two N×N arrays to a single N×2N array. Figure 2(ii) illustrates this mapping. For the sake of clarity, in this figure, only the interleaving of the elements in the first rows are shown explicitly; the remaining rows are interleaved in a similar manner.

3.1 Energy Savings due to Interleaving

Interleaving can reduce the energy consumption in the off-chip partitioned memory by increasing the effectiveness of low-power operating modes. As mentioned earlier, in the partitioned-memory architecture, only those memory modules containing the parts of the arrays currently being accessed need to be active. If we use the array interleaving strategy, it would colocate portions of different arrays which are accessed at the same time in a smaller number of modules. This can provide an opportunity for transitioning more modules into a low-power mode, and in most cases, keeping them in a low-power operating mode for a longer period of time.

3.2 Energy Savings due to Improved Locality

Interleaving can also reduce the number of accesses to the off-chip memory modules by enhancing spatial locality (cache locality) and can increase the inter-arrival times of off-chip memory accesses. This gives the compiler/hardware more opportunities for exploiting deeper sleep modes (more energy-saving operating modes) and/or keeping modules in low-power operating modes for longer periods of time.

Let us consider Example 1. In this example, if considered individually, each of the references has perfect spatial locality as successive iterations of the i loop access consecutive elements from each array. However, if, for example, the base addresses of these arrays happen to cause a conflict in the

Benchmark	Dataset Size	Number of Arrays
`biquad_n_sections (biquad)`	7 MB	six
`convolution (conv)`	8 MB	two
`fir`	8 MB	two
`lms`	8 MB	two
`n_complex_updates (complex)`	4 MB	four
`n_real_updates (real)`	8 MB	four
`fft`	7 MB	four
`eflux`	7 MB	eight

Figure 3. Benchmark codes used in our experiments and their important characteristics.

cache, the performance of this nest can degrade dramatically. The characteristic that leads to this potential degradation is that between two successive accesses to array U_1 there is an intervening access from U_2, and vice versa (which can distort locality). On the other hand, after interleaving, for a given loop iteration, two references (in a given iteration) access two consecutive array elements (much better locality).

3.3 Other Impacts of Interleaving

In addition to those listed above, array interleaving can have other benefits as well. Since it improves spatial locality, it helps to reduce the number of cache write-backs which, in turn, can reduce the energy spent in the cache memory itself. Improved spatial locality also has a positive impact on performance. It should be noted, however, that array interleaving makes array subscript calculations (address calculations) more complex; this may, in turn, depending on the capabilities of the back-end compiler, cause an increase in datapath (core) energy consumption. Also, in cases where one (or a small group) of interleaved arrays are accessed in a separate nest, the cache performance (and energy behavior) can suffer due to large strides.

4 Optimization Framework

We focus on array-dominated codes used extensively in image and video processing application domains. A common characteristic of these codes is that they manipulate arrays of signals using multiple nested loops, with array subscript expressions and loop bounds being affine (linear plus a constant term) functions of the enclosing loop indices and loop-independent variables. To interleave arrays in a given program, two subproblems need to be addressed. First, we need to identify the arrays to be interleaved. Second, we need to transform the program by replacing the original array accesses and declarations with their interleaved (transformed) counterparts.

We formulate the problem of selecting the arrays to be interleaved on an undirected graph called array transition graph (ATG). In a given ATG(V, E), V represents the set of array variables used in the program, and there is an edge $e = (v_1, v_2) \in E$ with a weight of $w(e)$ if and only if there are $w(e)$ transitions between the arrays represented by v_1 and v_2. A transition between v_1 and v_2 corresponds to the case when the array variable represented by v_2 is touched immediately after the array variable represented by v_1, or vice versa. Once the ATG has been built, our approach discovers the paths it contains, and interleaves the arrays that belong to the same path. The details of the algorithm to build an ATG and determine paths can be found elsewhere [5].

Once the arrays to be interleaved have been selected, they may still not be amenable to array interleaving transformation. This could be due to one of the following reasons:

- The arrays are not accessed with the same frequency in the innermost loop. Interleaving two arrays will be most useful when they are traversed within the innermost loop at the same speed. If, however, the subscript functions of two arrays do not contain exactly the same set of loop indices, one of them will be traversed faster than the other rendering interleaving difficult and not as effective.
- The arrays in question are of different dimensionality. Although, in principle, it is possible to interleave these arrays (e.g., by replicating the smaller array), this can bring about subtle coherence problems between replicas when the original array is updated.

Our current framework can interleave arrays with the same access frequency and dimensionality even if they differ in one or more different dimension sizes (extents). In such cases, the framework first determines the smallest rectilinear portions (from each array) that capture the simultaneously accessed array elements, and then interleaves only those portions. Having pruned the set of arrays to be interleaved, for the remaining arrays in each path, we use data transformations to interleave them [5].

5 Experiments

In this section, we evaluate the proposed interleaving optimization from energy as well as performance perspectives using eight array-dominated codes. Six of these codes (`biquad`, `conv`, `fir`, `lms`, `complex`, and `real`) are from the DSPstone benchmark suite [3]; `fft` is a two-dimensional Fast Fourier Transform code and `eflux` is an array-dominated benchmark code from the Perfect Club benchmarks. The important characteristics of these codes are given in Figure 3. Unless stated otherwise, our default bank (module) configuration is 4 × 2MB, i.e., four memory banks, each with a capacity of 2 megabytes. Further, all energy consumption values are those consumed in the DRAM memory

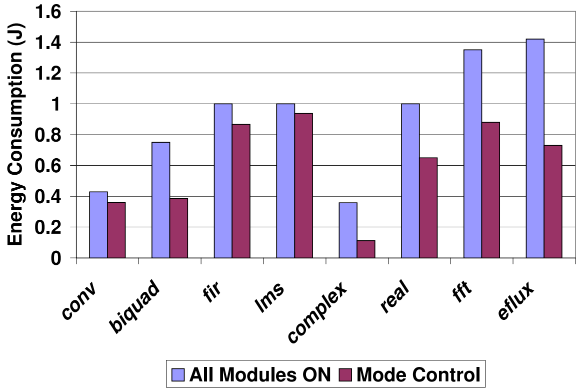

Figure 4. Energy impact of mode control (the `CTP` scheme) over a strategy in which all memory modules are fully active (ON) during the entire execution (cacheless system).

modules and do not include the (negligible amount of) energy consumed by the predictors that are part of the memory controller.

We first evaluate the energy benefits due to operating mode control only. Figure 4 gives the energy consumption in Joules (J) for two different versions of each code: (i) without mode control; that is, all memory modules are in active mode during the entire execution; and (ii) with mode control using the `CTP` strategy (without any cache memory). The energy consumption varies for each benchmark based on the number of memory operations executed. Further, we observe that using mode control is beneficial from energy perspective for all codes in our suite (an average of 24.6% saving in energy).

Figure 5 compares our three prediction mechanisms with and without array interleaving (again, for a cacheless system). In this graph, `CTP`, `ATP`, and `HBP` denote the cases where only the respective mode control mechanism (predictor) is activated. On the other hand, `CTP Opt`, `ATP Opt`, and `HBP Opt` denote the corresponding versions with array interleaving. We observe that array interleaving makes a large impact on energy behavior of all three mode control mechanisms. The average percentage reduction brought about by interleaving is 54.2% for the different configurations. Also, as far as the versions without array interleaving are concerned, the `HBP` version outperforms the rest. However, except for a few cases, using array interleaving with `CTP` or `ATP` results in better energy consumption than obtained through `HBP` without array interleaving. Considering that the `HBP` requires a more complex hardware mechanism, we can tradeoff this complexity using a less sophisticated hardware and array interleaving. In the rest of the paper, we consider only `HBP` and `CTP` schemes as they represent two extremes in sophistication.

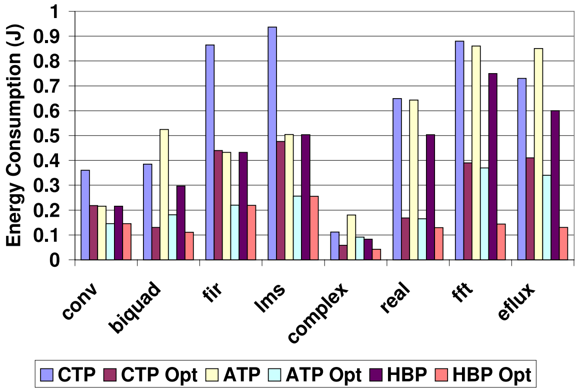

Figure 5. Comparison of different mode control mechanisms (cacheless system).

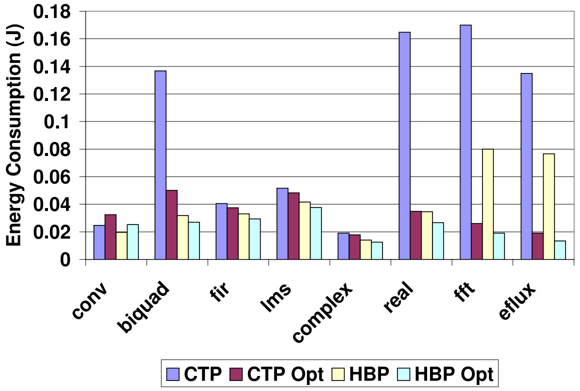

Figure 6. Comparison of `CTP` and `HBP` (with a 32KB, 4-way set associative cache).

Figure 6 shows the energy consumption resulting from `CTP` and `HBP` with and without array interleaving for a 4 × 2MB memory configuration with a 4-way data cache of 32KB. We see that, as compared to a cacheless system, the effectiveness of array interleaving here is reduced. On the average, array interleaving improves the energy consumption over mode control by 15.3%. The largest improvement occurs with `fft`, which is more than 80% for `CTP`. The reason for this reduced effectiveness is the fact that the existence of a cache memory reduces the number of accesses to the memory banks, allowing more memory banks to be placed into a low power mode, and for longer periods of time. This increased effectiveness of mode control diminishes the additional improvements due to array interleaving. We also observe that array interleaving causes an increase in energy consumption with `conv` as the miss rate of this code slightly increases after array interleaving.

We also investigated the impact of using mode control

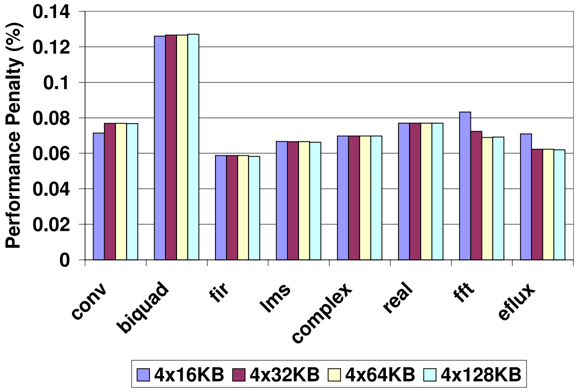

Figure 7. Performance impact of the mode control using `CTP`.

on performance. Figure 7 shows the performance penalty (increased percentage of cycles due to transition from low-power mode to active mode) when using `CTP` with different cache sizes. It must be noted that for the experimented set of benchmarks, the miss rate reduction due to interleaving was not significant enough to overcome the mode control performance penalty.

6 Conclusions

This paper presents an automatic mechanism for array interleaving to save energy in partitioned (multi-banked) memory architectures with power control features. This automatic mechanism is developed on top of a mathematical framework in which array access patterns of an application are captured using a graph-based representation (ATG), and transformed using linear one-to-one data transformation matrices. We validate the effectiveness of this transformation mechanism using a set of DSP benchmarks, and observe energy savings across different memory configurations and mode control mechanisms.

References

[1] 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.

[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 International Conference on High Performance Computer Architecture*, Monterrey, Mexico, January 2001.

[3] DSPstone Benchmark Suite. http://www.ert.rwth-aachen.de/Projekte/Tools/ DSPSTONE/dspstone.html.

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

[5] M. Kandemir. Array unification: a locality optimization technique. In Proc. *International Conference on Compiler Construction,* ETAPS'2001, Genova, Italy, 2-6 April, 2001.

[6] R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power-aware page allocation. In Proc. *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* November 2000.

[7] S. Y. Liao. *Code generation and optimization for embedded digital signal processors.* Ph.D. Thesis, Dept. of EECS, MIT, Cambridge, Massachusetts, June 1996.

[8] Rambus Inc. http://www.rambus.com/.

[9] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[10] W-T. Shiue and C. Chakrabarti, Memory exploration for low power, embedded systems, *CLPE-TR-9-1999-20, Technical Report,* Center for Low Power Electronics, Arizona State University, 1999.

[11] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* June 2000.

[12] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM Symposium on Programming Languages Design and Implementation*, pages 30–44, June 1991.

[13] M. Wolfe. *High-performance compilers for parallel computing*, Addison-Wesley Publishing Company, 1996.