Using Data Compression to Increase Energy Savings in Multi-bank Memories*

M. Kandemir[1], O. Ozturk[1], M.J. Irwin[1], and I. Kolcu[2]

[1] The Pennsylvania State University, University Park, PA, USA
{kandemir,ozturk,mji}@cse.psu.edu
[2] UMIST Manchester, M601QD, UK
ikolcu@umist.ac.uk

Abstract. New DRAM technologies such as SDRAMs, RDRAMs, EDRAMs, CDRAMs and others are vying to be the next standard in DRAMs and improve upon bandwidth limit of conventional DRAMs. With proliferation of power-aware systems, banked DRAM architecture has emerged as a promising candidate for reducing power. Prior work on optimizing applications in a banked memory environment has exclusively focused on uncompressed data. While this may be preferable from a performance viewpoint, it is not necessarily the best strategy as far as memory space utilization is considered. This is because compressing data in memory may reduce the number of memory banks it occupies and this, in turn, may enable a better use of low-power operating modes. In this paper, we explore the possibility of compressing infrequently used data for increasing effectiveness of low-power operating modes in banked DRAMs. Our experiments with five highly parallel array-based embedded applications indicate significant savings in memory energy over a technique that exploits low-power modes but does not use data compression/decompression.

1 Introduction

Low power dissipation in portable battery-operated platforms has drawn significant interest in the past decade. In many applications targeting at embedded platforms, a large fraction of energy consumption is due to main memory accesses. Recent work [1, 2, 3, 5] has suggested multi-banking as a way of reducing memory energy consumption. The main rationale behind this approach is that per access energy consumption is proportional to the size of the memory, and a small memory bank consumes much less (per access) energy than a large monolithic structure. In addition, unused memory banks in a multi-bank architecture can be placed into low-power operating modes to further energy savings.

Prior work on optimizing applications in a banked memory environment has exclusively focused on uncompressed data. In other words, the data manipulated by the application have been kept in memory in an uncompressed form throughout the execution. While this may be preferable from a performance viewpoint, it is not necessarily the best option as far as memory space utilization is considered. This is because compressing data in memory may reduce the number of memory banks it occupies

* This research is partly supported by NSF Career Award #0093082.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 310–317, 2004.

and this, in turn, may enable a better use of existing low-power operating modes (i.e., unused banks can be placed into low-power operating modes).

It is to be noted, however, that there are several important issues that need to be addressed before one has a reasonable data compression strategy.

- How should the available memory space be divided between compressed and uncompressed data?
- What should be the granularity of compression? A whole array, a data page, etc.?
- Which compression strategy should be employed when a given data block is to be compressed/decompressed?

The third question posed above is orthogonal to the ideas explored in this paper. That is, the proposed strategy can work in conjunction with any data compression/decompression algorithm. Therefore, without loss of generality, in this paper we use the algorithm proposed by Kjelso et al [4]. In this work, we concentrate on the remaining questions and propose a strategy for optimizing the effectiveness of low-power operating modes. While prior work employed data compression for energy savings and performance improvement [7, 9], to the best of our knowledge, this is the first study that considers data compression within the context of banked memory architectures.

Our strategy divides the available memory space (memory banks) into two disjoint parts (groups): one that holds compressed data and one that holds uncompressed data. The main objective here is to keep non-hot data (i.e., the data that are not very frequently used) in the compressed form to the extent possible. Another important issue here is to cluster the compressed data as much as possible for increasing the effectiveness of low-power modes. The timing for compressions/decompressions is also an important issue. Typically, one does not want to access data while it is in the compressed form (since it needs to be decompressed before it can be used; and this requires extra cycles in execution). In addition, when a new data is created or an old data is re-written, one needs to decide whether to store it in compressed or uncompressed form. In the rest of this paper, we focus on a banked memory architecture, and evaluate several compression-based memory bank management strategies for making best use of available low-power operating modes in a DRAM-based memory architecture.

This paper is structured as follows. The next section gives an overview of banked memory architecture and low-power operating modes. Section 3 presents details of our compression-based memory management strategy. Section 4 gives experimental data. Section 5 concludes the paper by summarizing our major results, and giving a brief discussion of ongoing work.

2 Banked Memory Architecture and Low-Power Operating Modes

We target a memory system that contains a number of banks, each of which can be energy-controlled independently. To save energy in this architecture, we put unused memory banks into a low-power operating mode. We assume the existence of three operating modes for a memory bank: active, napping, and power-down. Each mode is characterized by its power consumption per cycle and the time that it takes to transition back to the active mode (referred to as the resynchronization time or resynchronization cost). Typically, lower the energy consumption, higher the resynchronization

time. These modes are characterized by varying degrees of the bank components being active. Table 1 shows our operating modes, their per cycle power consumptions, and resynchronization costs.

Table 1. Operating-modes for memory banks.

	Energy Consump tion (nJ/cycle)	Resynchronization Cost (cycles)
active	3.570	0
napping	0.320	30
power-down	0.005	9,000

A memory controller that interfaces with the memory bus controls DRAM banks. The interface is not only for latching the data and addresses, but also to control the configuration and operation of the individual banks as well as their operating modes. For example, programming a specific control register in each memory bank could do the operating mode setting. Next is the issue of how the memory controller can be told to transition the operating modes of the individual banks. In this paper, we use a hardware-based approach to control mode transitions. In this approach, there is a watchdog hardware that monitors ongoing memory transactions. It contains some prediction mechanism to estimate the time until the next access to a memory bank and circuitry to ask the memory controller to initiate mode transitions. The specific hardware depends on the prediction mechanism that is implemented. In this work, we use an *adaptive next-mode prediction scheme*. In this scheme, if a memory bank has not been accessed for a while, then it is assumed that it will not be needed in the near future. A threshold is used to determine the idleness of a bank after which it is transitioned to a lower energy mode. The threshold is adaptive in the sense that it tries to adjust for any mispredictions it has made. Specifically, it starts with an initial threshold, and transitions to the lower energy mode if the bank is not accessed within this period. If the next access is to come soon after that (the resynchronization energy consumption is more dominant than the savings due to the lower energy mode), making the mode transition more energy consuming than if we had not transitioned at all, the threshold is doubled for the next interval. On the other hand, if we find that the next access comes fairly late, and we were overly conservative in the threshold value, then the threshold is reset to the initial value (one could potentially try more sophisticated techniques such as halving the threshold as well). Our objective in this work is to demonstrate that using data compression, one can increase the effectiveness of this low-power mode management scheme; i.e., we can either put more banks into low-power modes, or we can use more aggressive (i.e., more energy-saving) low-power mode for banks.

3 Compression-Based Memory Management

We explore design space in a systematic way, focusing on each design decision separately.

Memory Space Division. There are two different ways of dividing a given memory space between compressed and uncompressed data. In the “static” strategy, p out of a total of m memory banks are reserved for compressed data, whereas the remaining banks are allocated for uncompressed data. Obviously, if most of the data manipu-

lated by the application are not hot (i.e, not frequently accessed), one may want to increase p. On the other hand, in the "dynamic" strategy, the number of memory banks allocated to compressed and uncompressed data changes dynamically during the course of execution. Note that the dynamic strategy is expected to perform better than the static one if the behavior of the application changes during the course of execution.

Compression Granularity. Data can be compressed/decompressed in different granularities. In the "fixed granularity" scheme, we divide each dataset (array) into blocks of fixed size (e.g., 4KB), and the block sizes for all datasets are the same. Alternately, in the "variable granularity" scheme, each dataset (e.g., an array) can have a different block size (though all the blocks belonging to a given dataset are of the same size). Note that, as a special case of this latter scheme, one can set the block size of each dataset to the dataset size (in which case we have only 1 block per array).

Data Creation/Re-writing Strategy. When the data is created or re-written, we have flexibility of storing it in compressed or uncompressed form. In the rest of this paper, these two strategies are termed as the "compressed store" and "uncompressed store." Ideally, this decision should be made considering the future use of the data in question. It should be noticed that it is also possible to make this decision by analyzing the data access pattern exhibited by the application. More specifically, if the compiler can determine that a data block will not be used for a long period of time, it can be stored in the compressed format. On the other hand, if it will be reused shortly, it can be stored in an uncompressed form. This *compiler-directed* strategy is called "dynamic" in the remainder of this paper, since it tunes the store strategy based on the reuse of each data block.

Decompression Strategy. Another important question is when to decompress data. One obvious choice is "on-demand" decompression, whereby data is decompressed (if it is in the compressed format) only when it is really needed. In contrast, in "pre-decompression," data is decompressed before it is actually accessed. This latter alternative is only possible when we have extra cycles to pre-decompress data. Also, this is a compiler-based strategy in that the compiler analyzes the code, identifies the potential reuses of each data block, and inserts explicit pre-decompress calls in the code to initiate pre-decompression of data blocks.

Based on these, Table 2 shows our four-dimensional design space, a subset of which is explored in this paper from both energy and performance (execution cycles) perspectives.

Let us now discuss the implementation details of these different memory management strategies. Static memory space division is easy to implement. Basically, we select a value for p, and use p banks for the compressed data. However, implementing dynamic strategy is more involved. This is because we need to decide when to change the value of p during the course of execution. To achieve this, we make use the concept of the *miscompression rate* (or MCR for short), which is the fraction of the data accesses where the data is found in the compressed format in memory. Obviously, MCR should be reduced as much as possible. Based on MCR, our dynamic memory space division strategy operates as follows. If the current MCR is larger than a high-threshold (high-MCR) we reduce p, and if it is lower than a low-threshold (low-MRC), we increase p. We do not change p as long as the current MCR value is between low-MCR and high-MCR. In other words, as long as we are operating between

Table 2. Possible strategies for compression/decompression based memory bank management.

Method Id	Space Division	Comp. Gran.	Rewriting Strategy	Decomp. Strategy
1	static	fixed	dynamic	on-demand
2	static	fixed	dynamic	pre-decomp
3	static	fixed	uncompressed	on-demand
4	static	fixed	uncompressed	pre-decomp
5	static	variable	dynamic	on-demand
6	static	variable	dynamic	pre-decomp
7	static	variable	uncompressed	on-demand
8	static	variable	uncompressed	pre-decomp
9	dynamic	fixed	dynamic	on-demand
10	dynamic	fixed	dynamic	pre-decomp
11	dynamic	fixed	uncompressed	on-demand
12	dynamic	fixed	uncompressed	pre-decomp
13	dynamic	variable	dynamic	on-demand
14	dynamic	variable	dynamic	pre-decomp
15	dynamic	variable	uncompressed	on-demand
16	dynamic	variable	uncompressed	pre-decomp

low-MCR and high-MCR, we assume that the current division of the memory space is performing well. Also note that the current MCR value needs to be updated at regular intervals. In our current implementation, this is achieved as follows. Each time a block is accessed, we check whether it is compressed or not. If it is compressed, we increase a counter. At regular intervals (whose period is programmer-tunable), we compute the MCR, and change the memory space partitioning between compressed and uncompressed data if necessary.

As far as the compression granularity is concerned, implementing variable granularity is relatively easy if the block size for a given dataset is set to its total size (i.e., block size = array size). In this case, only bookkeeping necessary is a data structure that keeps track of which datasets are currently in the compressed form and which ones are not. However, implementing the fixed block strategy is more challenging, as we need to keep track of the status of each data block individually. In order to do this, our implementation employs a bit map, where each bit represents the status of a block. Consequently, each memory access goes through this bit map to determine the status of the data block to be accessed. In this paper, we experimented with only two re-writing strategies: uncompressed store and dynamic. We did not consider the compressed store strategy as our initial experiments showed that its performance is extremely poor (since, due to temporal locality of data, it is not a good idea to store the data always in the compressed format). Comparing the two strategies we experimented with, it is easy to see that the dynamic strategy requires extra compiler support (which will be explained shortly) to measure the reuse distance of the data (whereas the static strategy does not need such help).

Finally, while implementing on-demand decompression strategy is not very difficult, to implement the pre-decompression strategy, *the compiler needs to analyze the program and schedule data decompressions ahead of the time.* The necessary com-

Table 3. Default simulation parameters used for in experiments.

Simulation Parameter	Value
m	8
p	3
(High-MCF,Low-MCF)	(30%, 5%)
Block Size	2KB

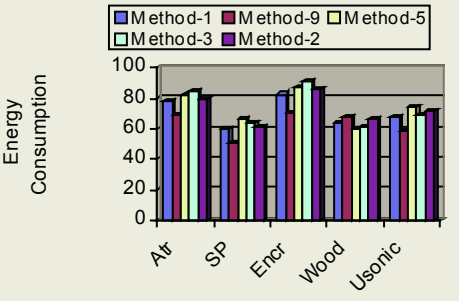

Fig. 1. Normalized energy consumption values with different methods.

piler support for this is similar to that of software-based data prefetching [11]. It is to be noted that this pre-decompression scheme can be successful only when we have extra cycles to pre-decompress the data. In our current implementation, we use an extra (pre-decompression) thread to perform pre-decompressions. This thread shares the same resources with the main execution thread. Whenever there are idle cycles, it kicks in and performs necessary (soon to be needed) decompressions. In the worst case, it may happen that the main execution thread wants to access a block and finds it compressed (which means that the pre-decompression thread could not find opportunity to decompress the block in question). If this happens, we proceed by allowing the pre-decompression thread to finish its job before the block is accessed.

In this work, we implement and study a subset of the strategies listed in Table 2. There are two main reasons for that. First, some of the strategies in Table 2 do not make much sense. Second, by being a bit careful in selecting the subset mentioned, one may have a good insight on the trends of interest as far as energy and performance behaviors are concerned.

4 Experimental Evaluation

Table 3 lists the default parameters used in our simulations. We use 4-issue processor core with 8K instruction and data caches. Later in our experiments we modify some of the parameters given in this table to conduct a sensitivity analysis. Recall that Table 1 gives the characteristics of the low-power operating modes used in this work.

All energy numbers presented in this paper have been obtained using a custom memory energy simulator (built upon SimpleScalar 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 compression-based strategy on memory system energy.

All necessary code restructurings for implementing “dynamic” and “pre-decompression” strategies have been implemented within the SUIF framework from Stanford University [8]. To implement the dynamic strategy, we keep track of data reuse

information at the array block granularity (instead of array element granularity as done in conventional compiler-based locality optimization studies). For this purpose, we employ the proposed solution by Wolf and Lam [10], and feed the data block information to the compiler. Implementing pre-decompression is very similar to implementing compiler-directed data prefetching. Basically, the compiler predicts future use of data and inserts appropriate pre-compression calls (instead of prefetching calls) in the code.

Table 4. Our benchmark codes.

Bench	Brief Description	Numof Lines	Energy Consumption (mJ)
Atr	Network address ranslation	626	13.38mJ
SP	All-nodes shortest path algorithm	1028	29.31mJ
Encr	Digital signature for security	1411	36.06mJ
Wood	Color-based surface inspection	978	22.80mJ
Usonic	Feature-based estimation	1005	37.44mJ

We test the effectiveness of our optimization strategy using five benchmark codes. Table 4 presents descriptions and important characteristics of these benchmarks. The last column in this table gives memory energy consumption when no compression/decompression is employed but available low-power operating modes are fully exploited. While we also have access pointer-based implementations of these benchmark codes, in this study we used array-based implementations.

We focus mainly on five different methods (from Table 2), and Figure 1 gives their memory energy consumption values, normalized with respect to the case without compression (i.e., with respect to the last column of Table 4). The reason why we are focusing on these five methods can be explained as follows. We consider Method 1 as the base and obtain the other versions by changing one parameter at a time (see the column titles in Table 2). One can observe from Figure 1 that the average energy improvement brought about by Method 1 is 29.78%, indicating that our compression-based scheme can be very successful in practice. Now, if consider Method 9, we can see from Figure 1 that its average saving is significantly better than that of Method 1 (36.66%). To understand this better, we focus on one of our applications (Usonic) and present its bank usage behavior. The x-axis in the graph in Figure 2 represents the time, and the y-axis gives the total size of the compressed data (on a time quantum basis) in terms of the number of banks it occupies. For example, if the y-axis value is 4.3 at a given time quantum, it means that the total size of the compressed data can occupy 4.3 banks. One can conclude from this graph that the amount of bank space

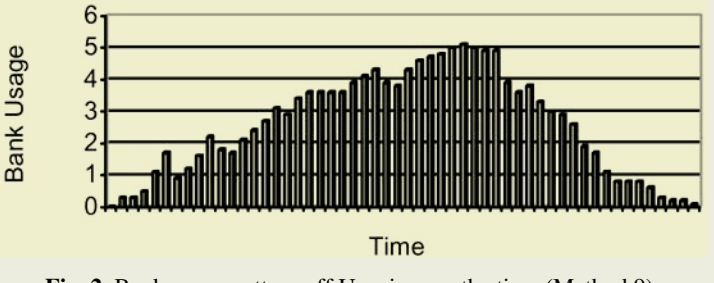

Fig. 2. Bank usage pattern off Usonic over the time (Method 9).

occupied for the compressed data varies over the time. Therefore, a strategy (such as Method 1), which allocates 3 banks for the compressed data (i.e., p=3 as shown in Table 3) throughout the execution, cannot generate the best energy behavior. In other words, for the best behavior, the amount of space allocated for the compressed data should be varied. In fact, we can see from Figure 1 that dynamic space division is beneficial for all applications except one.

5 Ongoing Work and Concluding Remarks

Banked-memory architectures enable power savings through low-power operating modes. While previous compiler, OS, and hardware based techniques showed significant savings in memory energy consumption by making use of these low-power modes, this paper demonstrated that further savings are possible if one employs data compression. We first described the potential search space, and then evaluated some select implementation methods. Our ongoing research includes experimenting with more sophisticated strategies (such as the one that allows different block sizes for different arrays), and evaluating the impact of our approach on multi-programmed workloads.

References

1. 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.
2. 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.
3. 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,*, 2002.
4. M. Kjelso, M. Gooch, and S. Jones. Performance Evaluation of Computer Architectures with Main Memory Data Compression. Elsevier Science, *Journal of Systems Architecture*, 45 (1999), pp. 571-590.
5. A. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power-Aware Page Allocation. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
6. 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.
7. B. Abali et al. Memory Expansion Technology (MXT): Software support and performance. *IBM Journal of Research and Development,* Vol 45, No 2, 2001.
8. 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.
9. L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors. In *Proc. DATE,* 2002.
10. M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation,* June, 1991.
11. T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems,* 16(1):55-92, February 1998.