External Memory Page Remapping for Embedded Multimedia Systems

Ke Ning
Analog Devices Inc.
3 Technology Way
Norwood, MA 02062
ke.ning@analog.com

David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115
kaeli@ece.neu.edu

Abstract

As memory speeds and bus capacitances continue to rise, external memory bus power will make up an increasing portion of the total system power budget for system-on-a-chip embedded systems. Both hardware and software approaches can be explored to balance the power/performance tradeoff associated with the external memory.

In this paper we present a hardware-based, programmable external memory page remapping mechanism which can significantly improve performance and decrease the power budget due to external memory bus accesses. Our approach was developed by studying common data access patterns present in embedded multimedia applications. In the paper, we evaluate a mechanism that can perform page remapping of external memory. We also develop an efficient algorithm to map application data and instruction memory into external memory pages. We employ graph-coloring techniques to guide the page mapping procedure. The objective is to avoid page misses by remapping conflicting pages to different memory banks (i.e., by assigning them different colors). Our algorithm can significantly reduce the memory page miss rate by 70-80% on average. For a 4-bank SDRAM memory system, we reduced external memory access time by 12.6% . The proposed algorithm can reduce power consumption in majority of the benchmarks, averaged by 13.2% of power reduction. Combining the effects of both power and delay, our algorithm can benefit significantly to the total energy cost.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles

General Terms Algorithms, Design

Keywords memory page remapping, memory coloring, embedded systems, memory controllers

1. Introduction

When designing a low-power embedded system, one key challenge is to design an architecture that can supply the functional units with sufficient data while also considering the associated power consumption. External memory is typically a challenging cost/performance design point in low cost, power-constrained, embedded systems due to their associated latencies and high bus capacitance.

Multimedia embedded software applications (such as MPEG-2/H.264 encoding/decoding, JPEG encoding/decoding, speech processing G.721, and PGP encryption algorithm) possess high memory locality. These applications are typically characterized by intensive data memory access, along with highly regular access patterns.

It has been observed that an access-pattern-aware cache and memory layout techniques can be used to improve the performance of external memory access [2]. However, most approaches have been focused on improving performance. There has been very little work that considers associated power impact.

In this paper, we first present a characterization of external memory access patterns presented in multimedia workload. Then, we describe our data-driven strategy to optimize the memory access speed and power consumption in a multi-banked memory system. We also describe an efficient coloring algorithm to best allocate memory banks. Finally, our experimental results indicate that our proposed scheme is very effective in improving bus performance and reducing memory system power/energy.

2. Related work

Past research has shown that embedded applications exhibit deterministic data access patterns. Since memory delay is a substantial portion of program execution, we need to effectively exploit this knowledge of data access patterns to tune embedded applications.

In many embedded multimedia applications, execution is dominated by loop bodies that cycle through arrays or matrices of data elements. Loop transformations (e.g., loop unrolling, loop tiling) and array restructuring can improve data access and performance by increasing the locality associated with these memory accesses [4]. Previous work has shown that we can reorder memory accesses to significantly improve application performance [8, 11].

Some common ways to characterize the external memory access patterns associated with multimedia data are stride and inter-reference gap [14]. Brown et al. [3] characterized memory access patterns using a tool that could capture the number of open pages, number of accesses between each reference to the same page, and a number of related statistics. More recently, Lee [10] also studied the performance and power tradeoffs in embedded multimedia applications. He reported that multimedia applications possess three types of memory access patterns. Based on his finding, he used prediction-based prefetching techniques to exploit the regularity in access patterns.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'07 June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006. . . $5.00

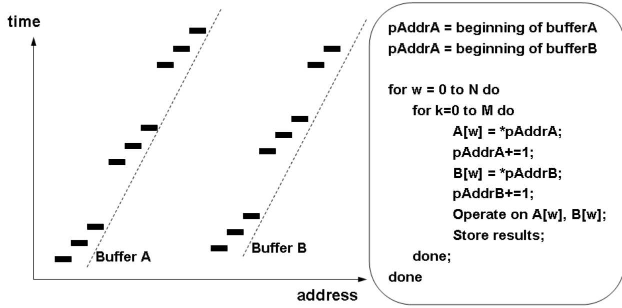

Figure 1. 2-way stream data access pattern and pseudo code of associated algorithms.

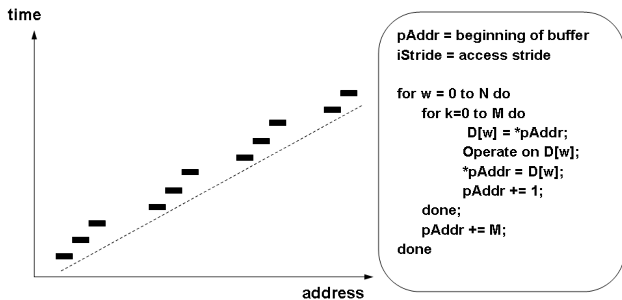

Figure 2. 2-D stride data access pattern and pseudo code of associated algorithms

Memory access pattern analysis has also been used to guide source code transformations and data layout. The goal of data layout optimization is to reduce the number of cache misses. Previous research looked at how to reduce conflict misses to improve performance and reduce power [5, 15]. De La Luz [6] showed that optimizations that target reducing off-chip memory energy can generate very different binaries and power budgets from those that target cache locality only. De La Luz used a memory system model wherein a subset of memory banks can be controlled individually and be placed in low power operating mode. Their compiler-based strategy modifies the original execution order of loop iterations in array-dominated applications to increase the length of the time period in which memory banks are idle. Their design allows them to control the operating mode for a given memory bank. They reported that their scheme can reduce the memory system energy expended by as much as 36.8% over a strategy that only uses low-power mode. Lebeck et al. [9] described OS-based strategies for optimizing energy consumption for a banked memory architecture. The idea is to perform page allocation in an energy-efficient manner by taking into account the banked structure of the main memory.

Pettis and Hansen [13] proposed a methodology for remapping instruction address space. They constructed a graph, where vertices corresponded to basic blocks. An edge is the weight between two vertices whenever the corresponding code blocks followed directly in sequential execution. The Pettis and Hansen algorithm attempts to place procedures that frequently call one another close together in memory in order to improve cache performance. Hashemi and Kaeli [7] proposed a color mapping memory layout algorithm to map program procedures, such that they can be cached more efficiently. They took into consideration the procedure size and cache organization. They showed that the instruction cache rate can be reduced by 40% over the original mapping and by 17% over the mapping algorithm of Pettis and Hansen.

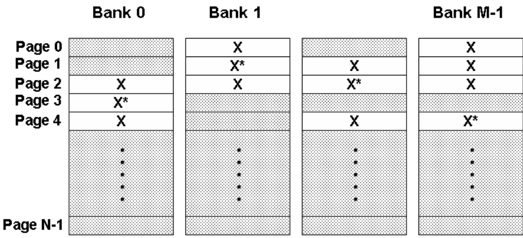

Figure 3. Data memory page layout for a memory system of four banks. An "X" indicates that the corresponding page is in use. An "*" indicates that the page is open.

One challenging aspect of the problem we are addressing in this paper is that we target external memory. All data accesses to external memory are already filtered by the on-chip cache hierarchy and further impacted by the direct memory access (DMA) engine. They also include both code and data accesses (though data references dominate, since embedded programs typically can be cached effectively on-chip).

In multimedia applications, the data access pattern includes several large scale buffers to store video frames, audio samples, and bitstream jitter buffers. The accesses to these buffers produce a large volume of non-contiguous accesses possessing large stride distances. Two dominant patterns are shown in Figures 1 and 2. Figure 1 presents a pattern commonly found in FIR digital filters and 2-D image blending. The 2-D stride access pattern shown in Figure 2 is commonly found in motion estimation, image preprocessing/post-processing, motion compensation and pixel rendering.

Our proposed solution has several advantages over previous work. First, previous solutions required access to the original source code, while our approach does not. Second, previous approaches limited their optimization to complete data entity boundaries (buffers, arrays, etc.) or entire code functions. From our analysis of data access patterns presented in multimedia applications, the majority of the execution time is spent on processing large monolithic data entities in embedded programs. Few of the previous approaches addressed optimizing access within individual data entities. Our proposed solution can break through that boundary and demonstrate the value of considering sub-entity optimization.

3. External Memory Pages

The main focus of our research is to improve the efficiency of access to multi-banked external synchronous DRAM (SDRAM), which is commonly used in embedded systems. We can think of a SDRAM memory system as a group of banks containing a large number of pages. The page size depends on the specific memory manufacturer and configuration. Typically, the size of a page is 1024 bytes (1 KB). A diagram of a typical synchronous DRAM (SDRAM) is shown in Figure 3. Inside each bank, only one page can be opened or accessed at a time. The maximum number of open pages is equal to the total number of memory banks.

When we first issue a memory reference to the memory subsystem, there is no memory page open and the memory controller must issue a full memory address (both row and column addresses) to the selected DRAM bank. Subsequent references to the same page (with the same row address) can be accessed by simply providing the column address, if the memory controller keeps the current row address active. This method allows lower access latency to the same page than would be possible if the controller had to provide a full address for each access. Any page which is accessed will remain open until a different page in the same bank is accessed. The

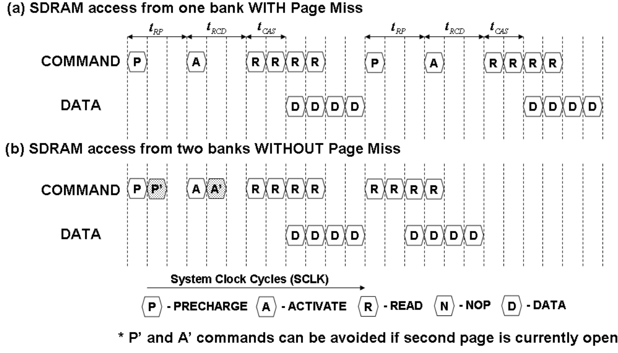

Figure 4. SDRAM access signal diagram. Two consecutive data reads are issued to SDRAM. In case (a), two reads are from same memory bank but different pages, which is a page miss situation. In case (b), two reads are from two different banks. If the second page is currently open, PRECHARGE and ACTIVATE commands are not needed.

currently opened page is forced to close in order to precharge the new row address strobe. This situation is referred to as an *external memory page miss*.

In a multi-banked memory system, if consecutive page accesses are to the same bank, the memory controller has to issue a PRECHARGE command to close the open page in that bank, followed by a ACTIVATE command to open the new page to be accessed. During the PRECHARGE and ACTIVATE cycles, the memory is only handling state transitions, and there is no actual data being transferred. This transition time is called the *page miss penalty*. If the next page access is in a different bank, the associated PRECHARGE and ACTIVATE can be overlapped with current ongoing data transfers. Therefore, the page miss penalty can hidden by the current transfer.

Page misses are major power and performance issues for SDRAM accesses due to the extra SDRAM command overhead. The page miss ratio is a useful metric to quantify the percentage of memory accesses that involve page miss operations. This ratio is computed as the number of times a page had to be opened divided by the total number of DRAM accesses. The more DRAM banks in a memory subsystem, the greater number of possible open pages. Figure 4 shows a 2-way stream data access with and without the page miss overhead. We can clearly identify the delay and power penalty imposed by the extra PRECHARGE and ACTIVATE commands, which are issued over the external bus when a page miss occurs. In the example of Figure 4, a page miss penalty can introduce an overhead of $t_{RP} + t_{RCD}$ external bus cycles. The external bus power is highly dependent on the memory technology chosen and it is closely proportional to the number of bus pins that toggled during each transaction. A page miss can cause more pin toggling due to the extra bus commands, but the size of the penalty depends on the state of the external bus.

4. Page Remapping Algorithm

Minimizing the page miss ratio is beneficial to both power and performance. We can reduce this ratio by reordering isolated data accesses into a single page or by using some memory layout strategies that can place sequential, cross-page, accesses possessing similar access patterns into different banks. The first approach has been investigated previously by either using dynamic allocation strategies in the operating system or by using compiler-based strategies. However, those approaches may not achieve the maximum benefit for multimedia applications wherein the dominant activities are within large monolithic data buffers possessing deterministic reference patterns. Those buffers generally contain many memory pages, but are viewed as single entities by a compiler or an operating system, and generally cannot be further optimized. Working at a page level, we can guide the hardware to tune accesses and break through the data buffer boundary. Our page level strategy is not limited to data only. It tunes both data and instruction accesses to external memory.

4.1 Motivation of Page Remapping

Returning to Figure 3, an SDRAM can be broken down into M number of independent banks, each bank contains N number of pages. Each page can be represented as $I_{i,j}$, where i is the bank index and j is the page index inside that bank. For a given program, a profiling tool can generate statistics on the frequency of *page transitions* occurring, where we move from page $I_{i,j}$ to page $I_{p,q}$, which we called $S(I_{i,j}, I_{p,q})$.

Page remapping techniques are widely used in memory management systems of modern computer systems. Modern 32-bit processors support a physical address extension (PAE) mode that allows the hardware to address up to 64 GB of memory. However, external devices (e.g., DMAs) can address only the lower 4GB of memory. One way to solve this problem is to copy the data into the lower 4GB memory space for DMA operations. The overhead associated with this approach is significant because it involves expensive memory copying which can impose significant overhead. To avoid costly memory copies, some computer systems provide programmable hardware support that can be used to remap memory for data transfers using a separate memory management unit (MMU).

Operating system loaders can map new pages into memory by either selecting a pool of available pages or allocating memory linearly until memory runs out. If we can use page profile information to instruct the operating system to remap pages into new physical locations, we can optimize external memory access. This only requires a small modifications to the operating system and the linker.

Figure 5 shows an embedded system with data caches and I/O DMAs both accessing external memory space. A "REMAPPING" block is inserted before the external bus interface unit (EBIU). The function of the "REMAPPING" block is to provide a mapping function f_{remap} between two page indexes that belong to the same memory page space.

$$f_{remap} : \{I_{i,j} \in Pages\} \rightarrow \{I'_{p,q} \in Pages\} \qquad (1)$$

By assigning weights W_i and W_j, a mapping function f_{remap} can be found such that the page miss overhead is minimized. The mapping function is represented in the form of a mapping table, which we call the *page remapping table* (PMT).

One page remapping table (PMT) will be provided to the operating system or linker along with each executing program. The operating system loader will load the program into memory pages according to the information present in the PMT and the operating system memory management system will program the remapping block as shown in Figure 5. These remapping algorithms are low overhead, particularly since multimedia programs tend to execute for a long period of time and remapping only occurs when the program is first loaded.

4.2 Remapping Algorithm

In this section, we described our remapping algorithms to generate the page remapping table (PMT). The basic idea of this algorithm is to treat the external memory as a two-dimensional page array. The pages on the same row in the array share the same row address. The pages in the same column share the same bank. The purpose of the remapping process is to re-allocate a page's physical location, such

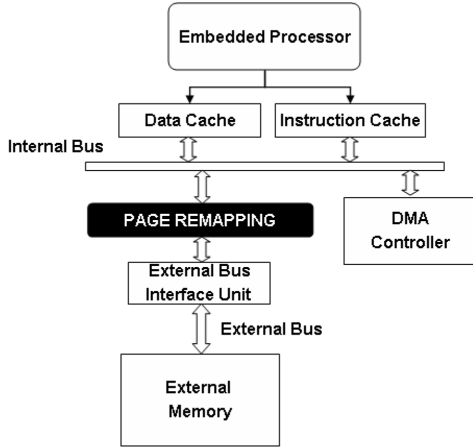

Figure 5. Memory page remapping system diagram. The "PAGE REMAPPING" block is the functional block to perform page remapping.

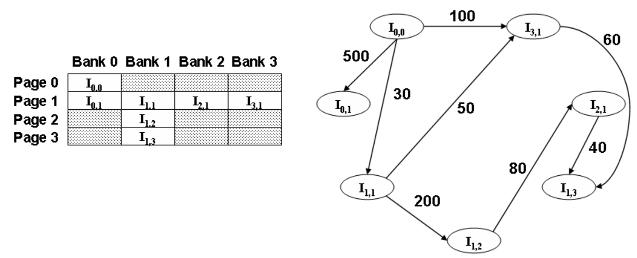

Figure 6. Example memory page layout and Page Transition Graph of a 4-bank SDRAM memory system. Clear blocks indicate occupied memory pages and gray blocks indicate unused memory pages.

that transitions within the same column or bank are minimized. In our algorithm, the remapped page is limited to reallocation within the same row. This restriction can drastically reduce the size of the PMT, because the remapped address will be larger than the size of a single memory bank. This has the added benefit of limiting cache aliasing due to our remapping since we are beyond the address range of typical on-chip caches.

The two dimensional nature of memory page layout allows us to define our algorithm as a graph coloring problem. Each bank can be represented as a distinct color. For each page, we first assign it a tentative color or bank. The process requires us to keep track of the total weight for the current assignment. If a remapped page conflicts with another page, we will then attempt to reassign colors to minimize the cost of a conflict. Our algorithm constructs a page transition graph. Each node represents a page, and the edge between two nodes is weighted with the number of times there is an immediate page transition between the two pages. Our algorithm only focuses on eliminating the first generation conflict between page transitions. A multi-step temporal relationship is the subject of future work.

When mapping a page to physical memory, our algorithm tries to avoid the transition penalties by assigning a color or bank which does not have any conflict or page transitions with previously mapped pages. If a conflict is inevitable, we will select the row of pages that causes the problem to follow a re-coloring process. By using the re-coloring process, we guarantee that the new mapping will not increase the overall weight by only resolving the first generation conflict in the transition graph.

4.2.1 Page Transition Graph

The transition graph captures page traversal in the program execution. Our algorithm starts by building a page transition graph with weighted edges. An edge weight is captured through the profiling processes and represents the number of times such a transition occurs. The heavier the weight associated with an edge, the more popular that page transition.

Figure 6 shows an example of the page transition graph based on the compiler-generated memory layout in the left side of the figure. We show a 4-bank memory system with 4 rows of memory pages, for a total of 16 memory pages. The program uses 7 out of 16 pages. To illustrate our algorithm, the 7 pages are arbitrarily distributed in memory. Each page is represented by its bank address and page row address. Page $I_{0,0}$ and $I_{0,1}$ are contiguous in physical memory. Page $I_{1,1}$, $I_{1,2}$ and $I_{1,3}$ are continuous in physical memory. Page $I_{2,1}$ and $I_{3,1}$ are non-contiguous and reside in banks 2 and 3, respectively. Non-contiguous memory layout is commonly encountered, because the compiler will place different data and instructions into different memory sections. These memory sections have predefined sizes in order to accommodate changes in the input data. Some memory is used as DMA buffers. Those memory sections tend to be located in a specific area in memory. The right side of Figure 6 is a page transition graph, which is generated using the profiling tool we have developed. The edge weights capture the page transition frequencies between any two pages during the entire program execution time.

4.2.2 Page Coloring

The job of page address remapping is to define a page remapping table (PMT) such that page transitions within SDRAM memory banks are minimized. We next describe the details of our page coloring algorithm and use the example in Figure 6 to demonstrate the procedure. To find an optimal page coloring is a NP complete problem, so we use heuristics to solve the problem.

The first step of the algorithm is to sort the edges based on their transition weights. Each weight represents the cost of transitions between two pages. We then perform mapping using this sorted set of edge weights and assign the pages connected by the edges to different colors. When we start the remapping process, no pages have been mapped. When processing each edge, we remap the pages associated with the edge. There are three types of information considered during mapping: 1) whether or not the page has been mapped, 2) the bank number for already assigned pages and 3) the weight parameter array. Every page has one unique weight parameter array, which contains the relationship of that page with every other page, and is organized on a per memory bank basis. It indicates the cost of putting one page into each of the memory banks. For example, if a page has a weight parameter array containing entries 500, 0, 100, 0, it means that the cost of mapping the page to bank 0 is 500 and bank 2 is 100, and there no cost for mapping to the two other banks. The goal of the algorithm is to minimize the final cost. Obviously in this case, the page should be mapped to either bank 1 or bank 3. All bank weight parameters are initialized to 0.

After a page is mapped, the parameter array for the mapped page and the page it connects to need to be updated. For all the edges being processed, there are five possible situations.

1. The first case occurs when neither of the two pages have been mapped yet into the physical memory and they are from different page rows, and there is an open slot in one of those two rows, but not in the same bank. This is the simplest case since we can basically assign them different colors and remap them to different bank numbers. The weight parameter array will be updated according to their bank locations.

2. The second case is similar to the first, except that one of the two pages is already mapped, and the unmapped page can be placed in a non-conflicting location.
3. The third case is similar to the first two, except that both pages are already mapped in separate banks. No remapping is needed, and only the weight parameters for those two pages need to be updated.
4. The fourth case occurs when an edge connecting two pages that have already been processed are mapped to the same row. They are guaranteed to be conflict free. There is no need to consider those transitions, so no action is needed.
5. The first three cases require that there are empty spaces in memory to allow pages to be placed in different banks. Case five is used to resolve the situation when that requirement does not hold. In this case, two pages can only be placed in the same bank (i.e., a conflict situation). We still place the pages in the only available banks, and then select one of the page rows to re-assign the bank number for all the pages on that row in order to resolve the conflict.

Each page has a weight parameter array indicating the cost of mapping it to each potential bank. Those weight parameters do not change and are independent of how pages are mapped. Only when the page is mapped into a bank that has a non-zero weight parameter, then that weight can be accounted for in the final cost. In case five, there are two options to minimize the conflict cost. One option is to apply an exhaustive search by iterating through all possible permutations of bank assignments and selecting the minimum cost. The complexity of this option is $O(N!)$ (N is the number of banks), and is probably suitable for a small number of banks (2 or 4). The other option is a greedy approach, which only swaps the conflicting page with one other mapped page. The complexity of this option is much lower ($O(N)$) and is suitable when a larger number of banks is used (8 or 16). In many cases, we will be able to produce a conflict free assignment. However, in some applications, conflicts can only be minimized.

The above process is repeated until all of the edges in the transition graph have been processed. Finally, after all the pages have been mapped into physical memory, the page re-mapping table (PMT) is generated according to the new page mapping. The PMT is used when a program is loaded on a target platform with page remapping enabled.

4.2.3 Example

In order to clearly describe the algorithm, we use the sample page transition graph in Figure 6 to demonstrate how to use our page coloring algorithm to remap pages in physical memory. After sorting all the edge weights in the graph in descending order, we have edge weights ranging from 500 to 30, and there are 8 distinct edges. The individual mapping step for each of the edges in the page transition graph is discussed in Figure 7.

There are eight edges in the graph. The algorithm starts with the heaviest edge in the transition graph, $I_{0,0} \rightarrow I_{0,1}$. Both page $I_{0,0}$ and page $I_{0,1}$ have not been mapped into memory yet, so they can be placed into the first available empty page slot in the memory as long as they are not in the same bank. We map $I_{0,0}$ into bank 0 and mark it as $I_{0,0}[0]$. We also map $I_{0,1}$ into bank 1 and mark it as $I_{0,1}[1]$. The parameter array for both pages is updated in order to record the mapping costs associated with all banks. Since $I_{0,0}[0]$ has a transition weight of 500, and $I_{0,1}[1]$ is currently mapped in bank 1, it means if page $I_{0,0}$ is mapped in bank 1 it will incur 500 page transitions, though it will incur zero cost if it is mapped into another bank. Therefore the weight parameter array for page $I_{0,0}[0]$ is updated from $\{0,0,0,0\}$ to $\{0,500,0,0\}$. Similarly, the weight parameter array for page $I_{0,1}[1]$ is set to $\{500,0,0,0\}$. Processing of edge 2 follows the same procedure as the first edge. We map the edge with the second largest weight (200), $I_{1,1} \rightarrow I_{1,2}$. Both nodes are mapped into the memory and their weight parameters are updated.

When processing the next three edges, the difference between these edges and the first two that have been processed are: one of the two pages that the edge connects has already been mapped into memory. Only the other unmapped page needs to be mapped. In step (3), edge $I_{0,0} \rightarrow I_{3,1}$ is processed. Page $I_{0,0}[0]$ is already mapped into bank 0. The only action needed is to find an open page slot in page row 1 and not in bank 0, because that would create a conflict with page $I_{0,0}[0]$. In that case, we map page $I_{3,1}[2]$ into bank 2 since it is available and then we update the parameter array to $\{0,500,0,0\}$ for page $I_{0,0}[0]$. Because $I_{3,1}[2]$ is in bank 2, its newly updated array becomes $\{0,500,100,0\}$. For the next two edges, we use the same process to map pages $I_{2,1}[3]$ and $I_{1,3}[0]$, respectively.

Next, edge $I_{1,1} \rightarrow I_{3,1}$ connects two pages which are both on page row number 1. No matter how we remap on that row, there will not be a conflict created between them. Therefore, no action is needed and no weight parameters need to be updated. This edge is processed using case 4.

The next edge $I_{2,1} \rightarrow I_{1,3}$ connects two pages which have both been previously mapped and have been mapped into different banks. No mapping action is needed in this case. The new edge does not affect any mapping, but the information it carries can be still useful and is reflected in the updated weight parameters. This edge is processed using case 3.

For the first 7 edges, we did not encounter a conflict. For every page that we were trying to map into the physical memory, there has been an open page slot available to assign the page to a different bank number (i.e., to a different color). No color conflicts occurred. In our experience, this is generally the case for most multimedia applications we have worked with. However, there may be situations where we can not find a conflict-free mapping. We have to design our algorithm to handle those cases as well, otherwise performance will suffer. For the last edge, we have a conflict situation. Edge $I_{0,0}[0] \rightarrow I_{1,1}[0]$ has both incident pages already mapped into bank 0. This is an example of case 3 above except that the two pages are conflicting. When such a conflict occurs, one of the page rows needs to be remapped in order to resolve the conflict. In practice, we choose the row that has a smaller number of mapped pages to resolve the conflict. In this example, it will be a lot easier to remap the page on page row number 0, because there is only one mapped page on that row. In order to illustrate the algorithm without making the example too complicated, we choose to use page row number 1 only for demonstration purpose. Page row number 1 contains four mapped pages $I_{1,1}[0]$, $I_{0,1}[1]$, $I_{3,1}[2]$ and $I_{2,1}[3]$, and each of them has a weight parameter array associated with it. In this example, since we use exhaustive search to find the page remapping permutation that produces the lowest transition cost. Figure 7 shows the results of the exhaustive search. A new mapping scheme for page row number 1 is generated, $I_{2,1}[0]$, $I_{3,1}[1]$, $I_{0,1}[2]$ and $I_{1,1}[3]$. In this example, the conflict has been resolved completely. Finally, the page weight parameter for $I_{0,0}[0]$ needs to be updated accordingly to reflect the location of the newly mapped page $I_{1,1}$. Since a conflict still exists and page $I_{1,1}[0]$ is mapped in bank 0, the weight parameter for $I_{0,0}[0]$ is $\{30,500,100,0\}$. After the remapping, page $I_{1,1}[3]$ is moved to bank 3, and the weight parameters for $I_{0,0}[0]$ become $\{0,500,100,30\}$

In case 5, conflict resolution is done by using the weight parameters, which does not guarantee that a conflict is resolved in an optimal way. Weight parameters are accumulated metrics of one page to all other pages. The remapping process will affect the weight

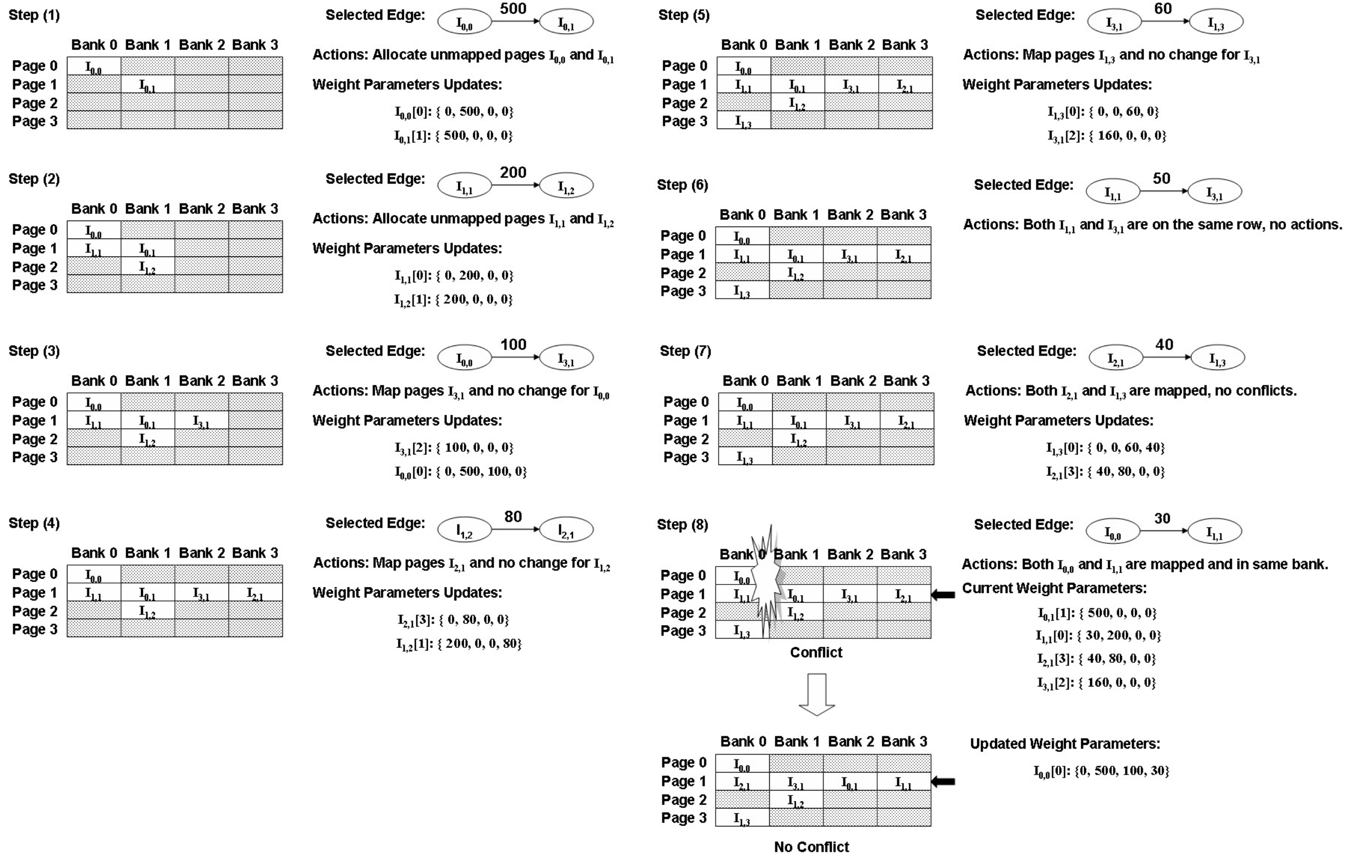

Figure 7. Our page remapping procedure using bank coloring. Each step contains the selected mapping edge, the memory page allocation and weight parameter updates.

Table 1. Benchmark used in the experiments and their important characteristics.

Benchmark	Total Cycle	Utilization Ratio	Request Per Cycle	Instruction Size	Data Size
MPEG2-ENC	82450027	0.377	0.0335	40 KB	4.5 MB
MPEG2-DEC	100000006	0.423	0.0348	41 KB	4.0 MB
H264-ENC	55621679	0.198	0.0162	63 KB	4.6 MB
H264-DEC	36106904	0.110	0.0150	32 KB	4.0 MB
JPEG-ENC	15410883	0.108	0.0079	19 KB	1.8 MB
JPEG-DEC	7666674	0.160	0.0110	18 KB	1.2 MB
PGP-ENC	462588015	0.002	0.0001	11 KB	0.6 MB
PGP-DEC	436908402	0.002	0.0001	11 KB	0.6 MB
G721-ENC	142895424	0.017	0.0015	16 KB	1.3 MB
G721-DEC	93717428	0.008	0.0023	16 KB	2.1 MB

$I_{0,0}$	00
$I_{0,1}$	10
	XX
	XX
	XX
$I_{1,1}$	11
$I_{1,2}$	01
$I_{1,3}$	00
	XX
$I_{2,1}$	00
	XX
	XX
	XX
$I_{3,1}$	01
	XX
	XX

Figure 8. Page remapping table (PMT) for example. ”00”,”01”, ”10” and ”11” are two bits information and ”XX” indicate undetermined values.

parameters for all other pages that have non-zero transitions in the other row which is being remapped. The remapping process will make future remapping for those pages potentially less optimal. The only redeeming grace is that conflicts tend to only occur for lightweight edges, and so should have a very low impact on the resulting performance and power.

At this point, all the pages have been mapped back to physical memory. The final mapping is shown in Figure 7. Each of the pages need only 2 bits (for 4 banks) to signal the new bank it is assigned to. The final page re-mapping table (PMT) is shown in Figure 8.

5. Proposed Architecture

The motivation of page remapping is to place consecutive page accesses into different memory banks. As a result, a large number of page misses can be absorbed or eliminated as shown in Figure 4, which can increase the memory bandwidth throughput, speedup program execution and decrease energy cost. The external memory can be viewed as a 3 dimensional space. The first dimension is the bank number, and translates into the first several most significant bits (MSBs) of the external memory address. It indicates to the memory controller which bank the current address is trying to access. The width of the bank address depends on the number of banks in the memory system.

The second dimension is the row address, and makes up the middle bits of the external memory address. It indicates to the memory controller which page inside that bank should be accessed. The number of bits required depends on the page and memory size. The last dimension is the column address. It is the address inside the accessed page. Typically, modern SDRAMs have a 1 KB page size. Some manufacturers still support 512 byte or even smaller. For example, if we have a 16 MB 4-bank SDRAM, the total number of bits needed to address that memory is 24 bits. Out of those 24 bits, the 2 MSBs are the bank address, the next 12 bits are the row address and the last 10 LSBs are the column address.

From the external memory address structure, we realize that by only remapping the bank address of every memory access, we can effectively distribute contiguous page accesses into different memory banks. The implementation of remapping blocks is tightly coupled with system memory architecture. Today there are mainly two types of embedded memory architectures: 1) systems without a virtual memory management unit (non-MMU) and 2) systems with a virtual memory management unit (MMU). The strategy that performs remapping in both types of systems are significantly different.

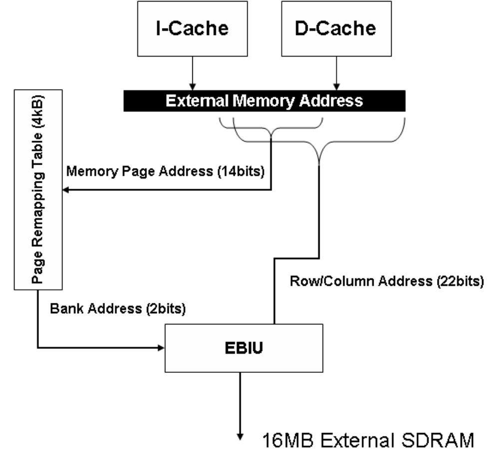

Figure 9. Bank address remapping block architecture. Memory in example is a 16 MByte 4-bank SDRAM.

5.1 Implementation Issues

In a non-MMU memory system, there is no virtual memory translation hardware. All of the memory references from caches or DMA are direct accesses. The memory space is linear and flat. The ”PAGE REMAPPING” logic has to be designed with the lowest implementation cost.

The function of page remapping is done inside the ”PAGE REMAPPING” block, and the whole process is transparent to the original software program. The hardware architecture to perform the bank address remapping is done through a look up table (LUT). Figure 9 shows an example of a page remapping block for a 16 MByte 4-bank SDRAM system. For every external memory address, the first 14 bits are extracted to be the index into the page remapping table (PMT). The entry inside the PMT is the new bank address (2 bits) for that particular page. For the given memory size and configuration, the size of the PMT is only 4 KB. If we use a 2-bank SDRAM, only 1 bit is needed for each page and for a 8-bank SDRAM memory, 3 bits are needed.

For a system that has a MMU memory system, the memory system itself already possesses an address translation mechanism. There is no need to insert more hardware into the system. We can take full advantage of the existing memory unit. In these systems, the page remapping table (PMT) becomes an page location requirement for an operating system loader. Before a program can be executed, the operating system has to allocate some number of open page slots (frames) in memory for the new program. Those page slots are allocated from a pool of pages which is managed by the operating system. Instead of treating all the pages the same (as the operating system usually does), our algorithm suggests that we keep N number of different pools of pages, where N is the number of memory banks of physical memory. Each pool contains the open pages from one memory bank. The operating system will need to allocate open pages from different memory pools according to the entry in page remapping table (PMT). For example, if the entry in PMT for page $I_{0,0}$ is ”10”, it means the operating system should allocate one open memory page for $I_{0,0}$ of a program from page pool of bank number 2(”10” = 2). It will provide the program with better memory efficiency.

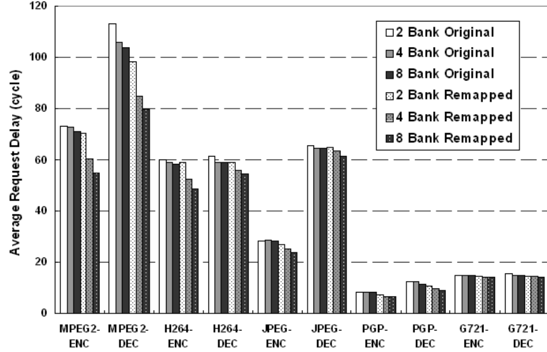

Figure 10. Average request delay in system bus clock cycle for multimedia benchmarks with 2-bank, 4-bank and 8-bank memory systems. Both original and page remapped programs are simulated.

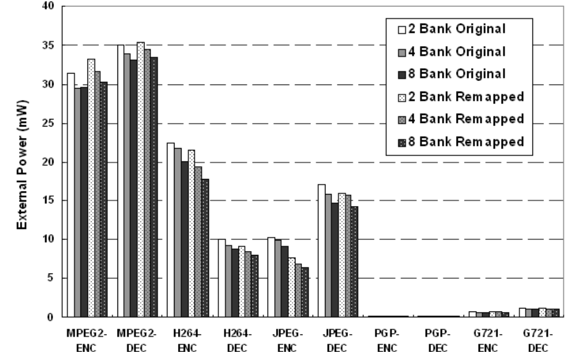

Figure 11. Average external bus and memory power for multimedia benchmarks with 2-bank, 4-bank and 8-bank memory systems. Both original and page remapped program are simulated.

6. Experiments

6.1 Methodology

In our experimental model, we chose the Analog Devices Blackfin family system-on-a-chip processors as our primary target. All the programs were compiled and simulated using the Analog Devices VDSP++ tool chain. We extended the simulator to generate a cycle accurate trace containing all external bus activities. We also added a module containing faithful simulation of our page remapping unit.

We start by first producing our page transition graph using a program trace profiler with page remapping disabled. The graph is fed into our page coloring algorithm to compute the optimal page remapping table (PMT). The generated page remapping table (PMT) is then reload back to the modified cycle accurate simulator with page remapping enabled. Our page remapping module will load the page remapping table (PMT) and filter all the address traffic going through the external bus. Both power and delay statistics are monitored. Cache models are also built into the simulator to accurately simulate the benchmark behavior, as the remapped external memory space could cause different cache behaviors. The results presented are based on an ADSP-BF533 processor model.

The goal of this study is to compare the power and performance benefits offered by our page remapping scheme. In our environment, we simulated an external SDRAM memory with 2,4 and 8 memory banks with a 1 KB page size. Our SDRAM model is modeled after a Micron 16 MB SDRAM. The detailed SDRAM configuration is described in more detail in an Analog Devices engineering note [1]. In our color mapping, the number of colors is equal to 2, 4 and 8, which is equal to the number of memory banks.

Table 1 describes the key attributes of the benchmarks we studied. Experiments were run on a set of multimedia workloads. We chose MPEG-2 and H.264 for video processing, JPEG for image compression, G.721 for voice processing and PGP for cryptography. All benchmark suites are representative and commonly used applications for multimedia processing. Utilization factor indicates how much load each benchmark places on external memory. It is calculated as the percentage of execution time when the external bus is busy. The embedded benchmarks we used do not contain dynamically allocated memory. All the data buffers present are statically mapped into physical memory. From the data in Table 1, we can see that the video benchmarks (MPEG-2 and H.264) tend to place more pressure on memory than the other benchmarks. PGP places the lightest load among all benchmark studied. The workload and memory sizes will affect the results of page remapping algorithms.

Table 2. Benchmark page miss reduction rate of 2-bank, 4-bank and 8-bank SDRAM memory systems.

Benchmark	2-BANK	4-BANK	8-BANK
MPEG2-ENC	8.3%	46.2%	66.5%
MPEG2-DEC	30.2%	54.5%	69.1%
H264-ENC	15.4%	60.7%	83.2%
H264-DEC	48.9%	79.9%	93.5%
JPEG-ENC	53.0%	76.3%	83.7%
JPEG-DEC	28.0%	48.1%	68.1%
PGP-ENC	70.0%	99.8%	100.0%
PGP-DEC	62.3%	93.0%	100.0%
G721-ENC	43.3%	72.0%	75.0%
G721-DEC	43.0%	66.7%	74.5%
Average	40.2%	69.7%	81.4%

We produced profiles of external memory bus activity using the timing/power model developed in our previous research [12]. The model allows us to simulate many processor-to-memory configurations accurately, and has been calibrated against actual hardware.

6.2 Results

6.2.1 Miss Rate

Table 2 compares the page miss rates for the original program and the page remapped program. We report the percentage improvement due to page remapping in SDRAM. The results show that by incorporating the page remapping module, we can achieve on average a 40.2% improvement on a 2-bank SDRAM system, 69.7% on 4-bank SDRAM system and 81.4% on a 8-bank SDRAM system.

Video benchmarks (MPEG-2 and H.264) and image processing benchmark (JPEG) are the primary targets for performance optimization due to their high memory demand and high power consumptioni. Table 2 shows that for video and image programs, the effect of page remapping steadily increases as the number of banks increases. For a 4-bank SDRAM, the page remapping can eliminate almost half of the page misses. For PGP encoding and decoding, page remapping can remove almost all of the page misses. The reason is that the PGP benchmark has a relatively small footprint, so page remapping can easily distribute the conflicting pages into different banks and resolve all the conflicts.

6.2.2 Power and Delay

To examine the impact of page remapping optimization on the performance of these benchmark programs, Figures 10 and 11 show the average delay and power for the original and page-remapped

Table 3. H.264 encoder benchmark results with two different input data sets: (i) CL and (ii) FB. A profile of CL input data set was used to construct the PMT in both cases.

SDRAM Banks	Experiment Results	CL			FB		
		Original	Remapped	Improve	Original	Remapped	Improve
2-BANK	Miss Rate (%)	37.1%	31.4%	15.4%	36.2%	31.6%	12.7%
	Delay (cycle)	59.9	59.1	1.3%	61.2	60.3	1.5%
	Power (mW)	22.4	21.5	4.0%	28.1	27.1	3.6%
4-BANK	Miss Rate (%)	34.9%	13.7%	60.7%	34.1%	14.1%	58.7%
	Delay (cycle)	59.1	52.4	11.4%	60.3	53.7	11.0%
	Power (mW)	21.7	19.4	10.7%	27.2	24.3	10.8%
8-BANK	Miss Rate (%)	29.8%	5.0%	83.2%	29.7%	5.7%	80.8%
	Delay (cycle)	58.3	48.5	16.8%	59.4	50.1	15.6%
	Power (mW)	20.1	17.7	11.9%	25.3	22.3	11.7%

Table 4. H.264 decoder benchmark results with two different input data sets: (i) tennis and (ii) carousel. A profile of tennis input data set was used to construct the PMT in both cases.

SDRAM Banks	Experiment Results	tennis			carousel		
		Original	Remapped	Improve	Original	Remapped	Improve
2-BANK	Miss Rate (%)	18.40%	9.40%	48.9%	22.20%	12.40%	44.1%
	Delay (cycle)	61.19	58.98	3.6%	67.08	64.72	3.5%
	Power (mW)	10.0	9.1	8.6%	12.5	11.4	8.4%
4-BANK	Miss Rate (%)	15.40%	3.10%	79.9%	19.40%	5.10%	73.7%
	Delay (cycle)	58.97	55.83	5.3%	64.89	61.62	5.0%
	Power (mW)	9.2	8.5	8.0%	11.6	10.8	7.0%
8-BANK	Miss Rate (%)	15.40%	1.00%	93.5%	19.40%	2.00%	89.7%
	Delay (cycle)	58.98	54.22	8.1%	64.91	59.51	8.3%
	Power (mW)	8.7	8.0	9.0%	10.9	10.0	8.6%

programs. For these results, the average delay is the number of external bus cycles to fulfill one external memory read or write request. The average power reported includes the external bus and memory power associated with the benchmark. Figure 10 indicates consistent results that our page remapping module can significantly reduce delay due to external bus requests. We can see a 17% speedup for MPEG2-ENC and a 20% speedup for MPEG2-DEC in a 4-bank SDRAM system. In an 8-bank memory system, the improvement is even higher. The speed improvements are direct results of page miss rate reduction. The more banks in a memory system, the more improvement our page remapping algorithm can produce. External bus power can be reduced if the signal toggling rate on the external bus decreases [12]. Figure 11 shows a mixture of results in terms of power savings. In the MPEG-2 benchmarks, the average power benefits are small. For other benchmarks, such as H.264 and JPEG, page remapped programs show significant power savings.

Reducing the number of page misses reduces the number of external bus commands and stall cycles, but may not lead to lower signal toggling per second. However, due to the reduced number of bus commands and stall cycles, the benchmark program can execute much faster, which will inevitably lead to lower energy consumption. Other benchmarks like H.264 and JPEG demonstrate both power and energy improvements. Their power improvements range from 4% to 30%.

6.3 Stability

To investigate the stability of the page remapping table (PMT) for different program inputs, we used two raw pixel sequence to drive the H.264 encoder. Both input sequences, CL and FB, have an image size of 720x480 (typically new data may arrive, but the image size will not change for a single program). The PMT table was generated for the CL sequence. Table 3 shows that the performance of FB is very close to the CL image. This is because multimedia programs possess a somewhat fixed data access pattern that is similar across different program inputs. Table 4 shows the performance of H.264 decoder benchmark with two different benchmark inputs. The results again are very close. We also tested the H.264 decoder for two different sets of inputs, tennis and carousel, as shown in Table 4. The PMT table was generated from tennis. Using the same PMT table, the performance of carousel is very close to tennis. This provides some evidence regarding the stability and resilience of our approach.

7. Conclusions

External memory system performance is critical for embedded applications. Previous approaches have focused on either compile-time or operating system implementations. The main contribution of this paper is to perform page remapping using a profile-guided hardware/software approach. Our algorithm takes into account the page access pattern of embedded multimedia applications. We describe an efficient coloring algorithm to intelligently map pages into the external memory in order to avoid page misses. Our experimental results demonstrate the benefits of our page remapping algorithm. We found that we can reduce the memory page miss rate by 70-80% on average over the original program mapping. The effect of reduced page miss rate translates to increased performance and reduced power consumption.

References

[1] Analog Devices Inc., Norwood, MA. *SDRAM Selection Guidelines and Configuration for ADI Processors*, May 2004.

[2] P. Bose, D. H. Albonesi, and D. Marculescu. Guest editors' introduction: Power and complexity aware design. *IEEE Micro*, 23(5):8–11, Sep/Oct 2003.

[3] M. Brown, R. M. Jenevein, and N. Ullah. Memory access pattern analysis. In *WWC '98: Proceedings of the Workload Characterization: Methodology and Case Studies*, page 105, Washington, DC, USA, 1998. IEEE Computer Society.

[4] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur. Predicting memory-access cost based on data-access patterns. In *CLUSTER '04:*

Proceedings of the 2004 IEEE International Conference on Cluster Computing, pages 327–336, Washington, DC, USA, 2004. IEEE Computer Society.

[5] F. Catthoor, E. D. Greef, and S. Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[6] V. De La Luz, I. Kadayif, M. Kandemir, and U. Sezer. Access pattern restructuring for memory energy. volume 15, pages 289–303, Piscataway, NJ, USA, 2004. IEEE Press.

[7] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 32, pages 171–182, New York, NY, USA, 1997. ACM Press.

[8] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. volume 48, pages 159–167, Washington, DC, USA, 1999. IEEE Computer Society.

[9] A. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOSIX)*, pages 105–116, Nov. 2000.

[10] J. Lee, C. Park, and S. Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 22–27, New York, NY, USA, 2003. ACM Press.

[11] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. volume 18, pages 424–453. ACM Press, July 1996.

[12] K. Ning and D. Kaeli. Power aware external bus arbitration for system-on-a-chip embedded systems. In *Proceedings of International Conference on High Performance Embedded Architectures and Compilers (HiPEAC05)*, 2005.

[13] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[14] V. Phalke and B. Gopinath. Program modelling via inter-reference gaps and applications. In *MASCOTS '95: Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 212–216, Washington, DC, USA, 1995. IEEE Computer Society.

[15] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of Principles of Programming Languages (POPL02)*, pages 140–153, 2002.