Co-Design of Interleaved Memory Systems

Hua Lin and Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{hualin, wolf}@ee.princeton.edu

ABSTRACT

Memory interleaving is a cost-efficient approach to increase bandwidth. Improving data access locality and reducing memory access conflicts are two important aspects to achieve high efficiency for interleaved memory. In this paper, we introduce a design framework that integrates these two optimizations, in order to find out minimal memory banks and channels required in the embedded system under performance restriction. Several important techniques, loop and data layout transformations for data access locality, extracting data streams, conflict cache miss reduction as well as data placement and optimally reordered access for interleaved memories, are incorporated in the design framework. Experiments show that our co-design method results in substantially less hardware requirement compared to the implementations without optimization.

Keywords

Interleaved memory systems, data access locality, memory access conflict, in-dimension-stride vector, extracted data stream, optimally reordered access.

1. INTRODUCTION

In an interleaved memory system, multiple memory banks are connected to a single bus (channel) and differentiated by the lower bits of the address bus (Figure 1). They share the bus by time division and overlapping the operations. Examples of interleaved memory systems can be found in the supercomputer area, such as CRAY X-MP [1], or by looking into the Rambus technology [2]. Interleaved memories achieve high bandwidth with reduced design cost, most significantly, the package pins, which is desirable for the cost-sensitive embedded systems where high data throughput are also required.

Design of an application specific interleaved memory system is a HW/SW co-design problem. Software design issues include data placement and access scheduling for data locality improvement and access conflict reduction. Hardware design issues include allocation of memory channels and banks and partitioning of the memory space. These hardware and software design decisions interact with each other. For example, increasing interleaving may require new data placement to reduce memory conflicts, whereas a new data placement may also inadvertently increase cache conflicts.

Memory access conflicts can severely impair the effective bandwidth of interleaved memories. When an access request is directed to a busy bank, it will be on hold until that bank is ready for new access. In the mean time, subsequent memory accesses are also hindered from being processed. Buffer designs [3, 4] may alleviate the bandwidth loss, however, the hardware cost is prohibiting. We consider no-buffer design in this work. To effectively utilize the bandwidth, memory access sequence needs to be evenly distributed across the memory banks, i.e. has good distributivity.

For a high performance memory system, cache and wide-word design are also indispensable factors. These features require good data locality which also tends to introduce sequential accesses to the memory banks. More reasons that prefer data locality could be: (1) Wide-word design is cheaper and more simple than interleaving to improve bandwidth; (2) Good data locality means less cache misses, less memory accesses and less possible conflicts. (3) For the cache with write-back policy, conflict cache miss will also cause memory access conflict (*dual conflicts*). This is because the number of cache line sets usually is no less than that of memory banks, thus the write back and fetch for the same cache line will also be mapped to the same memory bank. (Assume same line size, modular-2 address space. Write-through is not considered for its high bus traffic.) If the memory access has constant dual conflicts, the memory system will behave as if at most two memory banks were in working at any time. For these reasons, we optimize the data access locality before the distributivity optimization in our design framework.

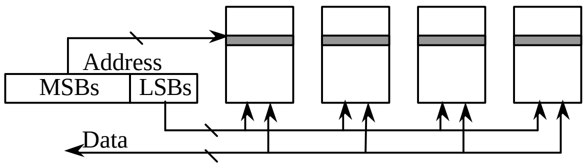


Figure 1: 4-way interleaved memory

Conflict analysis for vector streams [3, 5] has been well studied in supercomputer area. However, these analyses are mostly based on static vector stream model. In the real-time system, data streams are affected by program execution, cache behavior and access scheduling. Its dynamic characteristic should be considered and can be taken advantage of in the real-time system design. For example, because of the presence of cache, data access may not always go to the main memory. In the sample code in Figure 2, the data access for `x[i,j]` will be masked by that for `x[i,j+1]` because of group reuse. Also, the data access for a single reference `x[i,j+1]` will not always go to main memory because of self reuse. We use the term ***extracted data stream*** to refer to the data stream that actually reaches the main memory.

```
do i = 1, N-1
  do j = 1, N-1
    x[i,j]=0.5*(x[i+1,j]+x[i,j+1])
    y[i,j]=0.5*(y[i+1,j]+y[i,j+1])
enddo
```

Figure 2: An example

Assume the latency for a single memory bank is n_c clock cycles in this paper. If the number of memory banks in a channel is larger than n_c (*super-interleaving*), the maximal bandwidth of the channel will not increase, but the access conflicts may be reduced, so the effective bandwidth can be improved. When the number of memory banks in a channel reaches a certain large number, adding new channels to the memory system to increase bandwidth will be more cost-efficient. The requirement for memory channels and banks depends on the workload and how efficiently they can be utilized. In our design procedure, we start from an estimated minimal hardware configuration, and increase the number of memory banks or channels with corresponding data placement and access scheduling, until the execution time restriction is met.

In traditional memory design for embedded systems, interleaving has received little consideration. Wuytack et al. [6] studied the memory bandwidth minimization to achieve the minimal number of parallel memories or memory ports required. With interleaved memories, this problem could be quite different. To reduce memory conflicts, various methods [7, 8, 9, 10, 11] have been proposed. A new method, *optimally reordered access*, is proposed in this paper.

The next section introduces the architecture model for our work. Section 3 details our design framework, which covers the data locality and distributivity optimizations as well as the process to determine the minimal memory requirement. We show some experimental results and summarize our work in the last section.

2. ARCHITECTURE MODEL

In this work, we assume that data streams have relatively regular pattern and are tractable in computing the extracted data streams. Multimedia processing or wireless communications are examples of such applications.

A simplified architecture model is shown in Figure 3. The address generator works concurrently with the processor. When the processor is working on the current (nth) loop iteration, the address generator prefetches the data needed in the next ((n+1)th) iteration from main memory to cache, and selectively writes back the data produced in the previous ((n–1)th) iteration from cache to main memory. The selected cache lines for write back are those where conflict misses are predicted to happen later on. The loop iteration here can also be an *iteration packet*, a certain number of consecutive iterations, in order to achieve best scheduling result. The main memory could consist of multiple channels.

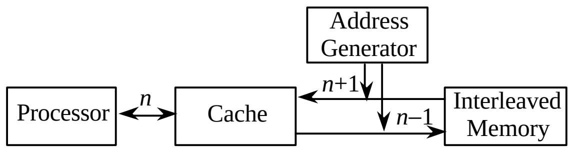


Figure 3: Architecture model

This model reduces the processor's waiting time by prefetching and selectively flushing the cache. The memory performance requirement is estimated by assuming load balancing in the data path, i.e. the data access time is equal to the data processing time. Data processing time is estimated by assuming zero waiting time for data retrieving to or from the memory. Selectively flushing the cache minimizes the chances of dual conflicts, which has been proved very effective to improve the memory performance by our experiments. Prefetching and flushing are statically scheduled by the compiler for the extracted data streams. Apparently, the estimation accuracy of the extracted data streams will affect that of the memory performance. The cache design incorporates necessary features, such as non-blocking for prefetching, flushing tag indicating a cache line's flushing status, etc. Obviously, the configuration of the cache architecture will have impact on the memory system performance. To focus on important facets of the main memory design, we assume the cache structure is fixed with suitable parameters in this paper.

The timing for the above scheduling is not absolute and may have overlapping. For example, when the processor is working on the nth iteration, the address generator may still be prefetching the data needed in the nth iteration. The purpose is to reduce the cache size required as long as the data dependence relationships are observed.

3. DESIGN FRAMEWORK

In this section, we first give the outline of our design framework, and then we step through important algorithms in the procedure.

3.1 Overview of the Design Framework

The design framework has the following steps:

(1) Loop and data layout transformations to improve the data access locality.

(2) Cache miss analysis to find the extracted data streams; data layout improvement to reduce conflict cache miss.

(3) Estimation of the lower bounds of the required memory channels and banks.

(4) Starting from the lower bounds, improve the data placement and schedule the data access to reduce memory conflicts, and see if the performance restriction is met. Increase the number of memory channels or banks if necessary until the performance restriction is met.

3.2 Definitions and Assumptions

The following definitions and symbols are used in this paper except otherwise indicated. Take the loop nest in Figure 2 as an example. Each iteration of the loop is indexed by a pair of values of the index variables *i* and *j*, which corresponds to a vector in a two-dimension space. We call it a *loop point*. The *iteration space* is the set of all the loop points. The indexing of the reference `x[i+1,j]` can be expressed as:

$$\begin{bmatrix} i+1 \\ j \end{bmatrix} = AI + c \ \text{ with } A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I = \begin{bmatrix} i \\ j \end{bmatrix}, c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where I is an iteration point, A is the *reference matrix*, and c is the *offset vector*. The *dimension size vector D* is the vector used to compute the address of an array element relative to the array's base address. For example, the address of x[i, j] is computed as $[i, j] \cdot D$ relative to the address of x[0, 0]

We assume that the array indexing and loop boundary expressions are affine, and reference matrices, offset vectors and iteration spaces are known at the compiling time. Because the code for a loop nest is usually small, instruction cache misses are few during the course of execution of the loop nest. Therefore, we ignore the instruction stream in the optimization procedure.

3.3 Improving Data Access Locality

To achieve data access locality, the innermost iterations of a loop nest should be accessing array elements through the fastest changing dimension of the array. Define *In-dimension Stride Vector* (ISV) as the distance vector between two loop points that access data in the same dimension of an array. The idea is as follows. If ISV exists, then: 1) Transform the layout of the array such that the dimension related to the ISV sits in the position of the fastest changing dimension which is the compiler's default configuration. 2) Transform the loop so that

$$Td = [0, 0, \ldots, 0, 1]^T \qquad \textbf{Eq. (1)}$$

where T is a transformation matrix, d is the ISV, i.e. the iterations related with the ISV become the iterations of the innermost loop.

Let's take a look at the sample code in Figure 2. For the reference `x[i,j]`'s first dimension, the ISV is [1, 0], i.e. two loop points with distance vector [1, 0] will access adjacent data in `x[i,j]`'s first dimension. Suppose the compiler uses row major. In order to utilize the ISV [1, 0] for data access locality, we need to switch the first and second dimension of array x, and also switch the outer and inner loop nest. Now consider the second dimension of array x, the corresponding ISV is [0, 1]. If we still assume row major, to utilize the ISV [0, 1], we do not need any transformation.

For two loop points I_1 and I_2 that access adjacent data in the r-th dimension of an array, we have ($\varnothing$: don't-care; superscript T: transpose; A^r: sub-matrix of A with r-th row removed):

$$(AI_1 + c) - (AI_2 + c) = [0, \ldots, 0, \varnothing, 0, \ldots, 0]^T$$
$$\Rightarrow \quad A(I_1 - I_2) = [0, \ldots, 0, \varnothing, 0, \ldots, 0]^T$$
$$\Rightarrow \quad A^r(I_1 - I_2) = 0$$

Lemma 1: The ISV space for the r-th dimension of an array with reference matrix A is the null space of A^r.

In our algorithm, all possible ISV's for each array reference are calculated and considered. Suitable transformations are chosen in consideration of the entire program for global optimization.

The ISV imposes partial restriction of the loop transformation matrix as shown in Eq. 1. In order to construct a valid transformation matrix, we also developed algorithms to test the existence of valid transformation matrix, and construct it if it exists. Please refer to our work [12] for details of the algorithms.

Combined loop and data layout transformations for data locality appeared in previous work [13, 14]. In Cierniak and Li's work [13], the method to search the data layout and transformation matrix was relatively weak. Kandemir et al. [14] gave an algorithm that was close to the idea of ISV. However, they still did not establish a systematic method to construct the valid loop transformation matrix.

3.4 Finding Extracted Data Streams

Extracted data streams are important for data access scheduling and memory bandwidth estimation. In the presence of cache, data access in the extracted streams consist of the memory bus traffic caused by cache misses. So the basic technique used here is cache miss analysis. The extracted data stream is different from the traffic caused by cache miss in that it is well recorded with its source, timing and attribute (read or write).

Since cache miss analysis has been extensively studied in Ghosh's work [15], we do not further discuss it here. Instead, we look at the example in Figure 2. If array x and y do not interfere with each other in the cache, and the cache is large enough to hold several rows of array x and y, we can expect that there is a write back stream for reference `x[i,j]`, a read stream for `x[i+1,j]` from the second row of array x, a read stream for `x[i,j+1]` for the first row of array x, and so to the references to array y.

After the cache miss analysis, data placement improvement will be applied if excessive conflict cache miss is found. This is also described in Ghosh's work [15].

3.5 Estimating the Minimal Configuration

Suppose the accesses in the extracted data streams are sent to the memory with no delay and processed with no conflict, the bandwidth required in this situation is a lower bound of the real configuration required. The lower bound is a starting point for the following interactive design procedure.

To estimate the lower bound, the critical part of the program (bus traffic vs. execution time) is chosen. Sum up the number of accesses in the extracted data streams for this part of program, and denote it as n_{ALL}. Suppose the execution time restriction is T_{PROC} memory clock cycles, then the lower bound for the required memory channels is

$$n_{\mathrm{ch}} = \lceil n_{\mathrm{ALL}} / T_{\mathrm{PROC}} \rceil \qquad \textbf{Eq. (2)}$$

If the lower bits of the address line are used to address the channels, n_{ch} is rounded to the next larger number that is a power of 2.

Now that the workload for each channel is known, the minimal number of banks in each channel can be determined by

$$n_{\mathrm{b}} = \left\lceil \frac{n_{\mathrm{ALL}} / n_{\mathrm{ch}}}{T_{\mathrm{PROC}}} n_{\mathrm{c}} \right\rceil \qquad \textbf{Eq. (3)}$$

and rounded to the next larger number that is a power of 2.

3.6 Reducing Access Conflicts

Changing data placement will change the destination memory banks to be visited. Reordering the access sequence will change the access time. Both are effective to reduce access conflicts. In our work, we apply both methods for better result.

3.6.1 Data Placement

Kurian et al. [11] proposed module partitioning and data placement method in their work. The address generation can be difficult in the implementation of their method. We use array padding, merging and address offsetting in our work. These methods are effective and more applicable.

For a single data stream, if it is not successfully optimized for data access locality, it will access main memory in each iteration. If the bank stride for the access stream is an odd number, the accesses will visit all the banks. If it is an even number, we change the dimension size of the array so that the bank stride changes to odd number.

Suppose the line width of the memory bank is w bytes. The bank stride s for consecutive accesses from a single reference is:

$$s = \left\lfloor \frac{D^T(AI_1 + c) - D^T(AI_2 + c)}{w} \right\rfloor = \left\lfloor \frac{D^T A(I_1 - I_2)}{w} \right\rfloor \qquad \textbf{Eq. (4)}$$

Here I_1 and I_2 are two consecutive iteration points. If both are inside the innermost loop, $I_1 - I_2 = [0, 0, \ldots 0, 1]^T$. The dimension size of the vector is the depth of the loop nests. At the boundary of the innermost loop, $I_1 - I_2 = [0, \ldots, 0, 1, I_L - I_H]^T$, where I_L and I_H are the lower and upper bounds of the innermost loop respectively. Boundary situation is only considered when $|I_L - I_H| << n_b$, where n_b is the number of memory banks.

If s is an even number, we need to find a minimal padding for the fastest changing dimension of the array, such that with the changed dimension size vector D_1, the following holds:

$$\frac{D_1^{\mathrm{T}} A(I_1 - I_2)}{w} = s + 1 \qquad \textbf{Eq. (5)}$$

For one-dimension array, the above padding is not applicable and more sophisticated padding method is needed. We do not further discuss it here.

For multiple data streams working concurrently, the following methods can be applied to reduce conflicts:

(1) Array merging. Two streams become a de facto single stream.

(2) Base address offsetting. For example, Denoting $z \rightarrow n$ as access from array z visiting bank n, by offsetting the base address of array y, the data stream
$x \rightarrow 1, y \rightarrow 1, x \rightarrow 2, y \rightarrow 2, x \rightarrow 3, y \rightarrow 3, x \rightarrow 4, y \rightarrow 4, \ldots$
can be changed to
$x \rightarrow 1, y \rightarrow 5, x \rightarrow 2, y \rightarrow 6, x \rightarrow 3, y \rightarrow 7, x \rightarrow 4, y \rightarrow 8, \ldots$
which has better distributivity.

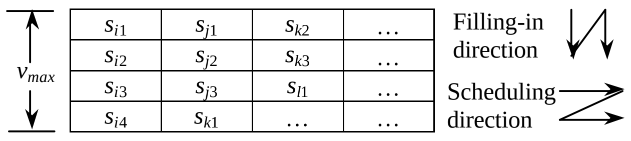
3.6.2 Optimally Reordered Access

Multiple streams can cause severe conflicts. Loop fission is a simple method to separate data streams where applicable. Corral [7] and Lee [8] studied methods to reorder the data access. Their methods are suitable for multiple vector streams that access the same subset of memory banks. The method *optimally reordered access* proposed here is effective for arbitrary data streams.

Let's see how fast it can achieve when the access order is re-arranged. Consider a single channel with n_b banks indexed from 0 to $n_b - 1$. For a data stream $\{s_i\}_{i=1}^{\infty}$ with a period length of p, where s_i is the bank number to be addressed, consider the section of one period S=$\{s_i\}_{i=1}^{p}$. The total number of accesses to bank i is denoted as v_i. Let $v_{\max}=\max\left(\{v_i\}_{i=0}^{n_b-1}\right)$, and denote $\mathrm{B}_{\max}$ as the set of banks that receive accesses for $v_{\max}$ times. We have:

Lemma 2: The minimal time to finish the accesses in the stream S is either p or $n_c \cdot v_{\max}$, whichever is larger.

Proof: Suppose t is the time used to finish the access sequence S. Apparently, $t \geq p$. Because the stream is periodic, and the minimal interval to access the same bank is n_c, $t \geq n_I \cdot v_{\max}$, or $t \geq \max(p, n_I \cdot v_{\max})$. Next, we construct a schedule to prove that the minimal time is achievable.



s_{i1}	s_{j1}	s_{k2}	…
s_{i2}	s_{j2}	s_{k3}	…
s_{i3}	s_{j3}	s_{l1}	…
s_{i4}	s_{k1}	…	…

Figure 4: Schedule table

As shown in figure 4, the schedule table has $v_{\max}$ rows and unspecified number of columns. Fill in the table the accesses in column major, from top to bottom and left to right, with the following rules: 1) Accesses to the same bank are filled in consecutively. 2) Accesses to the banks in the set $\mathrm{B}_{\max}$ are filled in first, followed by accesses to other banks.

Schedule the table in row major, from left to right and top to bottom with the following rule: if the number of accesses in a row $n \geq n_c$, then schedule all the accesses in this row; otherwise, schedule all the accesses in this row followed by $n_c - n$ time slots with no access request. We can see that there is no conflict in this schedule and it achieves the minimal time. The proof is straightforward and we skip it here. ♦

As an example, the following data stream
$x \rightarrow 1, y \rightarrow 1, x \rightarrow 2, z \rightarrow 1, x \rightarrow 3, y \rightarrow 3, x \rightarrow 4, z \rightarrow 3, \ldots$
can be reordered to
$x \rightarrow 1, x \rightarrow 2, x \rightarrow 3, x \rightarrow 4, y \rightarrow 1, y \rightarrow 3, z \rightarrow 1, z \rightarrow 3, \ldots$
for reduced conflicts.

The periodic attribute of the data stream is for simplicity of the proof so that the ending cycles of the access sequence do not complicate the proof of "minimal time". For sufficiently long sequence where the several ending cycles are negligible, the claim in Lemma 2 is practically true for non-periodic sequence.

From Lemma 2 we can easily prove: the minimal reordered access time $t_{\min}$ for a data stream S which is mixed from multiple streams S_i (i = 1, ..., k) will have $t_{\min}(\mathrm{S}) \leq \sum_{i=1}^{k} t_{\min}(\mathrm{S}_i)$. It is known that multiple streams are the main cause of conflicts. However, with reordered access, the situation is reversed because data access from multiple streams will have no worse distributivity than that of individual streams.

3.6.3 Implementation

The effectiveness of the reordered access depends on the distributivity of the accesses in the data stream. For this reason, in our design work, we first improve the distributivity of the data streams by data placement, then schedule the access with the reordered sequence.

Let's look at the same data stream as the above:
$x \rightarrow 1, y \rightarrow 1, x \rightarrow 2, z \rightarrow 1, x \rightarrow 3, y \rightarrow 3, x \rightarrow 4, z \rightarrow 3, \ldots$
We first offset array z's base address such that the accesses from array z: $z \rightarrow 1$, $z \rightarrow 3$ change to $z \rightarrow 2$, $z \rightarrow 4$. Then the data stream is reordered to
$x \rightarrow 1, x \rightarrow 2, x \rightarrow 3, x \rightarrow 4, y \rightarrow 1, z \rightarrow 2, y \rightarrow 3, z \rightarrow 4, \ldots$
which shows better result than a direct reordering.

Data placement is double checked to prevent inadvertently introducing conflict cache miss. When this happens, new placement is applied. The least common period of multiple streams is the unit for the reordered access scheduling, which corresponds to the iteration packet mentioned in section 2. If the packet length is too long for the cache to hold the prefetched data and not to evacuate useful data at the same time, we need to split one period into several sections and schedule each of them, and possible conflicts between sections should be counted. If there are multiple channels in the system, they can be configured as symmetrically parallel or interleaved. The reordered access scheduling is a direct extension from the proof of Lemma 2 for either configuration.

3.7 Allocating Channels and Banks

For a given memory configuration, Lemma 2 tells us the minimal data access time that can be achieved after applying reordered access. If the performance requirement is not met, we need to increase the number of memory channels or banks. At the same time, algorithms in section 3.6 that depend on the memory configuration will be reapplied.

Figure 5 shows the interactive procedure to determine the number of memory channels (n_{ch}) and banks (n_{b}). In the pseudo code, $n_{\max}$ is the maximal number of banks allowed in a channel, T_{PROC} is the the execution time restriction, and $t_{\min}(n_{\mathrm{b}}, n_{\mathrm{ch}})$ is the reordered data access time as a function of n_{b} and n_{ch}, which can be determined by Lemma 2. Start from the lower bound of (n_{b}, n_{ch}) estimated in subsection 3.5, do the following

```
while $t_{\min}(n_b, n_{ch}) > T_{PROC}$ do
    if $n_b < n_{\max}$ then
      $n_b \leftarrow 2n_b$
    else
      $n_b \leftarrow n_c$, $n_{ch} \leftarrow 2n_{ch}$  (or $n_{ch} \leftarrow n_{ch}+1$)*
    endif
enddo
```

* Depends on system configuration for multiple channels

Figure 5: Optimization procedure

If the number of memory banks in the channel reaches $n_{\max}$ and the execution time still cannot be met, new channels are added in order to increase the memory bandwidth. At the same time, the number of memory banks in a channel is reset to n_c, which is the minimal number to reach the full bandwidth of a channel.

4. EXPERIMENTS AND SUMMARY

In the experiments, we first ran some benchmarks to verify the effectiveness of the optimization procedure with known memory configuration. After that, we tested our co-design framework to find the minimal memory requirement under the execution time restriction.

The benchmark programs were compiled with DLX compiler, and the memory traces were sent to the trace-driven simulator DineroIII [16]. We modified DineroIII so that it can output the trace of the bus traffic. The memory access in the trace output is then assumed to be issued to the memory as fast as possible in order to measure the maximal memory performance achievable by a powerful processor.

In Figure 6, DCT is the fast DCT from MMSG [17] on a 352*288 image (2 bytes/pixel), FS is the full-search motion estimation on the same image, and MUL is a 256*256 (byte) matrix multiplication. The data cache is 16kB in size, 2-way associative and with a line size of 32 bytes. The main memory is single-channel with a latency of 4 clock cycles, n banks and 32 bytes in line size. The bars in the figure indicate relative performance.

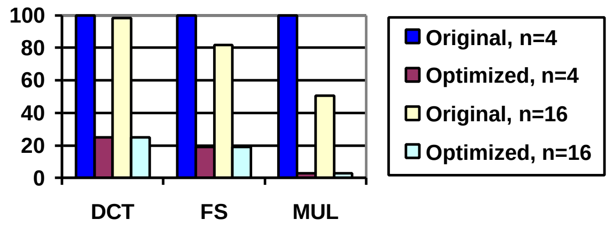


Figure 6: Performance comparison

DCT and FS work on memory blocks, thus have fairly good data locality. However, the results for the un-optimized (original) programs are not good. The trace output for DCT and FS shows that dual conflicts are responsible for about half of the performance loss. Another half comes from conflict cache miss which can be eliminated by proper data placement in the memory. The result also shows that, increasing the number of memory banks does not necessarily increase the performance that much without proper optimization. The un-optimized MUL shows poor memory performance because of poor data locality and layout.

Table 1 shows the result of the co-design procedure to find the minimal memory requirement when the program execution time is given. The first number in the pair is the number of memory channels required, the second is that of the memory banks. The result clearly shows the difference between a direct hardware implementation and a well-refined co-design.

	Cycle=32768			Cycle=65536		
	DCT	FS	MUL	DCT	FS	MUL
Un-optimized	2, 4	2, 8	157, 32	1, 4	1, 8	79, 32
Optimized	1, 2	1, 2	17, 4	1, 1	1, 1	9, 4

Table 1: Experimental result

Our work focused on locality and distributivity optimization for the memory access in order to achieve minimal memory cost. The framework left much room for further exploration. For example, the cache structure can be a variant in the design, or the scheduling with consideration of other resources in the system may improve the overall performance. Our framework could be a good starting point for further improvements in the co-design of interleaved memory systems.

5. ACKNOWLEDGMENTS

This work was sponsored by the National Science Foundation and the New Jersey Center for System-on-Chip Technology.

6. REFERENCES

[1] *CRAY X-MP Series Mainframe Reference Manual.* Cray Research Inc. HR-0032, Nov. 1982

[2] http://www.rambus.com/

[3] R. Raghavan, J. P. Hayes, "Reducing interference among vector accesses in interleaved memories", *IEEE Trans. Comput.*, Vol. 42, No. 4, pp. 471-483, Apr. 1993

[4] K. A. Robbins, S. Robbins, "Buffered banks in multi-processor systems", *IEEE Trans. Comput.*, Vol. 44, No. 4, pp. 518-529, Apr. 1995

[5] W. Oed, O. Lange, "On the effective bandwidth of interleaved memories in vector processor systems", *IEEE Trans. Comput.*, Vol. C-34, No. 10, pp. 949-957, Oct. 1985

[6] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, H. De Man, "Flow graph balancing for minimizing the required memory bandwidth", *Proc. IEEE 9th Int. Symp. System Synthesis (ISSS'96)*, La Jolla, CA, Nov. 1996, pp. 127-132.

[7] A.M. del Corral, J.M. Llaberia, "Minimizing conflicts between vector streams in interleaved memory systems", *IEEE Trans. Comput.*, Vol. 48, No. 4, pp.449-456, Apr. 1999

[8] D.L. Lee, "Memory access reordering in vector processors", *IEEE Symp. High Performance Comput. Architecture*, Raleigh, North Carolina, Jan. 1995, pp. 380-389

[9] D. H. Lawrie, "Access and alignment of data in an array processor", *IEEE Trans. Comput.*, Vol. C-24, pp. 1145-1155, Dec. 1975

[10] B. R. Rau, "Psudo-random interleaved memory", *Proc. 18th Int. Symp. Comput. Architecture*, Toronto, Canada, May 1991, pp. 74-83

[11] L. Kurian, B. Choi, P.T. Hulina, L.D. Coroor, "Module partitioning and interleaved data placement schemes to reduce conflicts in interleaved memories", *Proc. Int'l Conf. Parallel Processing*, Vol. 1, pp. 212-219, 1994

[12] http://www.ee.princeton.edu/~hualin/seminar.ppt

[13] M. Cierniak, W. Li, "Unifying data and control transformations for distributed shared memory machines", *Proc. SIGPLAN'95 Conf. Programming Language Design and Implementation*, Jun., 1995

[14] M. Kandemir, J. Ramanjujam, A. Choudhary, "Improving cache locality by a combination of loop and data transformations", *IEEE Trans. Comput.* Vol. 84, No. 2, pp. 159-167, Feb. 1999

[15] S. Ghosh, "Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behavior", Ph.D. Thesis, Dept. of Electrical Eng., Princeton University, Nov. 1999.

[16] M. Hill, A. Smith, "Evaluating associativity in CPU caches", *IEEE Trans. Comput.*, Vol. 38, No. 12, pp.1612-1630, Dec. 1989

[17] http://www.mpeg.org/MPEG/MSSG/