

# A Compiler Algorithm for Exploiting Page-Mode Memory Access in Embedded-DRAM Devices

Jaewook Shin, Jacqueline Chame and Mary W. Hall  
Information Sciences Institute  
University of Southern California  
{jaewook,jchame,mhall}@isi.edu

## ABSTRACT

This paper presents a compiler algorithm and several optimization techniques to exploit a DRAM memory characteristic (*page mode*) automatically. A page-mode memory access exploits a form of spatial locality, where the data item is in the same row of the memory buffer as the previous access. Thus, access time is reduced because the cost of row selection is eliminated. The algorithm increases frequency of page-mode accesses by reordering data accesses, grouping together accesses to the same memory row. We implemented this algorithm and present speedup results for four multimedia kernels ranging from 1.25 to 2.19 for a Processing-In-Memory (PIM) embedded DRAM device.

## 1. INTRODUCTION

Memory delays are a major performance bottleneck in embedded-DRAM systems, where the memory latencies seen by the processor are dominated by the on-chip-DRAM access time. DRAM modules support an efficient *page-mode* access, where a memory access to a location currently in the DRAM open-row buffer fetches the data directly from that buffer, eliminating the cost of fetching the row from the DRAM array. Page-mode accesses, when applicable, are supported by the DRAM's memory controller. To fully exploit lower latency page-mode accesses, the user or the compiler must reorganize the computation so that accesses to a same memory row are grouped together, and there are no intervening accesses to other rows.

In the past decade, most of the research on compiler optimizations for the memory hierarchy focused on exploiting data locality in caches [4, 7, 8, 9, 10, 15, 24, 25]. Although cache optimizations and page-mode optimizations have the common goal of exploiting data reuse (in caches or in the DRAM's open row, respectively), the analysis and code transformations required are different. For example, loop tiling is used to exploit temporal reuse in caches by bringing together in time loop iterations that access the same data. The goal is to keep the data accessed in a tile in cache, and the order of the accesses within a tile is not important. On the other hand, exploiting page-mode accesses requires not only bringing together in time loop iterations that access data in a same memory row, but also grouping these data accesses together. Exposing opportunities for grouping ac-

cesses to a same array may require transformations such as unroll-and-jam, to bring accesses issued in distinct loop iterations to the body of the transformed loop, and statement reordering, to group the memory accesses.

Recent research has proposed to exploit page-mode accesses through manual code transformations [19, 17, 3]. This paper presents a compiler algorithm for exploiting page-mode automatically. Our algorithm is implemented in the SUIF compiler infrastructure [13], and it leverages well-known compiler analyses and code transformations to identify potential page-mode accesses and group these memory accesses together. The algorithm is applicable to loop-based computations in general embedded systems and it is also applicable to embedded-DRAM systems designed to exploit the large on-chip bandwidths by transferring and processing objects larger than a machine word [23, 1].

We have performed an experimental evaluation of our algorithm on a Processing-In-Memory (PIM) device that is part of the DIVA architecture [12], where the PIM processor is capable of transferring and processing 256-bit objects (superwords) in parallel. Our results show the performance improvements from exploiting page-mode accesses, and the combined benefits of page-mode accesses and other compiler optimizations targeting architectures with support for *superword-level parallelism*<sup>1</sup> (SLP) [16, 22]. We obtain speedups ranging for 1.25 to 2.19 for four multimedia kernels. This paper makes the following contributions:

- A new compiler algorithm for automatically exploiting page-mode memory accesses;
- An experimental evaluation of the algorithm on four data-intensive multimedia kernels;
- A discussion of practical issues that must be addressed when exploiting page-mode accesses in combination with other compiler optimizations.

This paper is organized as follows. Section 2 motivates our approach using a simple example. Section 3 introduces our algorithm for exploiting page-mode memory accesses. Section 4 presents experimental results on a set of four multimedia kernels. Section 5 addresses practical issues which are the subject of future work. Related research is discussed in Section 6 and Section 7 concludes the paper.

## 2. MOTIVATION

Figure 1 illustrates the benefits of page-mode accesses using a simple loop nest with two array references. Assuming that

<sup>1</sup>Fine grain SIMD parallelism in a register larger than a machine word

Ref.	Loop j	Loop i
A[j][i]	$m * RMLatency$	$n * m * RMLatency$
B[i]	$m * RMLatency$	$n * m * RMLatency$
Total	$2 * n * m * RMLatency$	

(a) Original

Ref.	Loop j	Loop i'
A[j][i]	$m * RMLatency$	$\frac{n}{4} * m * RMLatency$
A[j][i+1]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
A[j][i+2]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
A[j][i+3]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i]	$m * RMLatency$	$\frac{n}{4} * m * RMLatency$
B[i+1]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i+2]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i+3]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
Total	$\frac{n}{2} * m * RMLatency + \frac{3n}{2} * m * PMLatency$	

(b) After unroll-and-jam and reordering

**Table 1: Memory Latency Computation**

the sizes of arrays  $A$  and  $B$  are larger than the DRAM’s open-row buffer, all array references in Figure 1(a) are in random-mode, since reference  $B[i]$  displaces the DRAM row containing  $A[j][i]$  from the open-row buffer and vice-versa.

For the same number of memory accesses in this loop nest, we can increase the page mode memory accesses by applying a series of code transformations, as shown in Figure 1(b). First, unroll-and-jam is used to unroll the outer  $i$  loop and fuse together the resulting inner  $j$  loop bodies. Unroll-and-jam creates opportunities for page-mode accesses by moving array references from successive loop iterations of the outer loop into the body of the transformed inner loop. Once the loop is unrolled and the copies of the loop body are fused, accesses to the same memory page in the loop body may be grouped together by reordering the memory accesses in the transformed loop body, if the reordering does not violate data dependences.

In Figure 1(b), following unroll-and-jam, where the  $i$  loop is unrolled by a factor of 4, references to the same array ( $A$  or  $B$ ) in the body of the transformed loop are grouped together. This results in page-mode accesses for all references in the loop body, except leading references  $A[j][i]$  and  $B[i]$ , which are in random mode.

Table 1 shows the total memory access cost for the code in Figures 1 (a) and (b), if we assume that latencies for random mode and page mode accesses are uniform, and that accesses are not going through cache. Assuming that random-mode latency is three times the page-mode latency as in [14], loop (a) has a total latency cost of  $6 * n * m * PMLatency$ , while (b) has a cost of  $3 * n * m * PMLatency$ , a factor of 2 difference in overall memory latency.

This example shows the potential for improving performance in embedded DRAM devices through the previously-described code transformations. To expose opportunities for page-mode accesses by applying unroll-and-jam and memory access reordering, a compiler algorithm must: (1) determine the safety of these code transformations and select a loop for which unrolling is profitable; (2) select an unroll factor that increases page-mode accesses while not causing register

```

for(i=0;i<n;i++){
  for(j=0;j<m;j++){
    load A[j][i]
    load B[i]
    ...
  }
}

```

(a) Original

```

for(i=0;i<n;i+=4){
  for(j=0;j<m;j++){
    load A[j][i]
    load A[j][i+1]
    load A[j][i+2]
    load A[j][i+3]
    load B[i]
    load B[i+1]
    load B[i+2]
    load B[i+3]
    ...
  }
}

```

(b) After unroll-and-jam and reordering

**Figure 1: Unroll-and-jam and Reordering**

spilling; and, (3) transform the code to reorder the memory accesses. In the next section we present our compiler algorithm for exploiting page-mode accesses, which includes the three steps above.

We have developed this algorithm in the context of a compiler for DIVA, a system-architecture that incorporates processing-in-memory embedded DRAM devices as smart-memory co-processors in an otherwise conventional system [12]. Although the proposed compiler algorithm is not specific to the requirements of the DIVA architecture, we describe the algorithm from the viewpoint of an architecture that supports *superword-level parallelism*, with an instruction set akin to multimedia extensions such as Intel’s SSE and Motorola’s AltiVec. Superword-level parallelism refers to performing the same operation in parallel on multiple fields of a superword, which is an aggregate object larger than a machine word. In the following algorithm description, we will refer to register width to support the notion that a machine might have different register widths for distinct objects. If a machine does not support superword-level operations, then the register width is the same as the machine word.

In previous work, we presented an algorithm for exploiting spatial and temporal locality in superword register files in a compiler that already supports superword-level parallelism [22]. In this paper, we show that with a similar approach we can also exploit spatial locality in the page of a DRAM memory array.

### 3. ALGORITHM

In this section we introduce a compiler algorithm for exploiting *page-mode memory accesses*. Our algorithm is applicable to loop nests with array references in the loop body, where the array subscript expressions are affine functions of the loop index variables. Only array accesses are reordered by the algorithm, since it is difficult to determine whether two scalar accesses are on the same memory page. For presentation purposes, we make some simplifying assumptions as

1. Select a loop to unroll
2. Control register pressure
3. Align the loop to page boundaries
4. Unroll-and-jam
5. Reorder memory accesses

**Figure 2: Algorithm**

follows.

1. Array objects are aligned at memory page boundaries.
2. The lowest dimension sizes of array objects are multiples of a memory page size.
3. The compiler backend does not change the memory access order generated by the algorithm.

Some of these assumptions can be removed by modifying the compiler backend (1,3) or by padding array objects (2).

The algorithm presented in this paper unrolls a single loop in a loop nest, since in practice unrolling more than one loop could create register pressure and instruction cache misses. A set of heuristics is used to select which loop to unroll and its unroll amount. These heuristics result in a fast algorithm that is effective for the benchmarks presented in Section 4.

However, unrolling multiple loops in a loop nest might expose more opportunities for page-mode accesses than when unrolling a single loop. In previous work [22] we present an algorithm for exploiting superword-level locality which uses unroll-and-jam to expose data reuse, and unrolls multiple loops in a nest. The computation of the unroll amounts requires a complex analysis to determine the exact number of superword registers needed to keep the data accessed in the loop. This complexity is due to several factors such as group reuse among copies of a reference created by unrolling (which may reuse data in superword registers) and self-spatial reuse of the original references.

A more complex algorithm for exploiting page-mode memory accesses which would consider multiple loops for unrolling is the subject of future work, and we plan to leverage our analysis and algorithm for selecting unroll amounts described in [22].

Figure 2 illustrates the steps of the algorithm, which are described in the remainder of this section. The first step selects which loop to unroll, after determining the safety of the code transformations (unroll-and-jam and statement reordering). The second steps selects an unroll factor that increases page-mode accesses while not causing register spilling. The last three steps apply the code transformations to the loop nest.

**Selecting a Loop To Unroll** The first step of the algorithm selects a loop to unroll, based on the number of random-mode memory accesses of the loop nest after applying unroll-and-jam. The algorithm uses data dependence information to determine the safety of unroll-and-jam and to prevent selection of unroll amounts greater than the dependence distance if inner loop dependence distances are negative.

For each loop  $l$  in the loop nest, the algorithm computes the unroll amount  $X_l$  and its corresponding number of random-mode accesses  $R_l$ , such that  $R_l$  is the smallest number of random-mode memory accesses if  $l$  is selected to be unrolled (assuming that references to a same memory page can be grouped together). Then the algorithm compares the number of random-mode accesses of each loop in the nest and selects the loop with the smallest  $R_l$ .

When computing the unroll amount  $X_l$  that minimizes  $R_l$ , the algorithm considers only references that are loop-variant with  $l$  in the lowest dimension. For a reference that is loop-variant with  $l$  in the lowest dimension, unrolling  $l$  and jamming the copies of  $l$  in the loop body creates opportunities for page-mode accesses between the copies of the original reference. On the other hand, unrolling loop  $l$  does not change the total number of random-mode accesses generated by references that are loop-variant with  $l$  in one of the higher dimensions. Loop-independent dependences can be removed by locality optimizations such as scalar replacement [2] or superword replacement [22].

For each loop  $l$  the smallest unroll amount that minimizes  $R_l$  is computed as in Equation 1.

$$X_l = \frac{P}{\min_{a \in A}(T(a) * C(a, l))} \quad (1)$$

where  $P$  is the memory page size,  $A$  is the set of array references in the loop nest which are loop-variant with  $l$  in the lowest dimension,  $a$  is an array reference in  $A$ ,  $T(a)$  is the type size of  $a$  and  $C(a, l)$  is the coefficient of the index variable  $l$  in the lowest-dimension subscript of  $a$ .

After computing the unroll amounts, the algorithm computes the corresponding number of random-mode memory accesses  $R_l$ , with the goal of selecting the loop with smallest  $R_l$ . For each loop  $l$ , the number of random-mode accesses  $R_l$  is computed as the number of distinct pages in the *memory-page footprint* of  $A$ ,  $F_l(A, X_l)$  (assuming that the algorithm can group together references to a same page). In previous work [22], we present the computation of the *superword footprint* of a set of array references in a loop nest, which consists of the number of distinct superwords accessed by the references, a function of the unroll amounts. The memory-page footprint can be computed in a similar way to that of the superword footprint. First, the set of references is partitioned into groups of *uniformly generated references* [25], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term<sup>2</sup>. Then, for each group of references, the algorithm computes the number of pages accessed in the unrolled loop body. Finally, the total number of pages is computed as the sum of those of each group of uniformly generated references.

**Controlling Register Pressure** After selecting a loop  $l$  to unroll, the algorithm adjusts the unroll amount of the selected loop to avoid register pressure and register spilling, which could offset the benefits of unroll-and-jam.

In a previous paper [22] we presented the computation of the number of registers required to keep the data accessed by the references in the loop nest after applying transformations for increasing locality in the superword register file. Here we present a simplification of this algorithm to provide the

<sup>2</sup>We assume that two or more references that access the same array but are not uniformly generated access distinct data in memory, which results in a conservative estimate of the number of memory pages.

intuition behind our approach.

We compute an upper bound of the total number of registers that can be simultaneously live by partitioning the references in the loop nest in groups of uniformly generated references and computing the superword footprint of each group.

For example, the number of registers required for a group that contains a single reference  $a$  that is variant with  $l$  is given by Equation 2, assuming  $C(a, l) = 1$ .

$$NR_l(a) = \frac{X_l \times T(a)}{W} \quad (2)$$

where  $W$  is the register width in bytes (for example,  $W = 4$  for a 32-bit scalar register, and  $W = 16$  for a 128-bit superword register such as the AltiVec's and  $T(a)$  is the type size of  $a$  in bytes. Equation 2.

The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The total number of registers required ( $TNR$ ) to keep the data accessed in the loop nest is computed as the sum of the number of registers required for each group of uniformly generated references. If the total number of registers is larger than the number of registers available, the algorithm adjusts the unroll amount  $X_l$ , by dividing it by the ratio of  $TNR$  and the number of available registers  $NREG$ .

$$X_l = \left\lfloor \frac{X_l}{\frac{TNR}{NREG}} \right\rfloor \quad (3)$$

The number of available registers  $NREG$  is given by number of registers in the register file minus the number of registers reserved by our algorithm for temporary storage.

Since the smallest type size is used in Equation 1, all references that have spatial reuse carried by loop  $l$  can exploit spatial reuse fully at the memory page level. Therefore, choosing a loop  $l$  that has the smallest random-mode accesses when unrolled by  $X_l$  is a reasonable choice. Dividing it evenly if too many registers are used, as in Equation 3, will result in a solution that is also aligned to a page boundary at the beginning of the loop. However, choosing a different loop can result in different register requirements. For example, if a loop is selected and then its unroll amount is reduced to half because of register pressure, there can be another loop that results in more random-mode accesses but requires fewer registers, leading to less overall random-mode accesses than the initial selection.

**Aligning the Loop To Page Boundaries** If the starting addresses of the memory accesses in the unrolled loop body are not aligned to a page boundary, each set of memory accesses to a same array will have one additional random-mode access per iteration. To remove these unnecessary random-mode accesses, step 4 of the algorithm splits the iteration space of the chosen loop into at most three loops (*head*, *body* and *tail*), so that the starting addresses in loop *body* are aligned to page boundaries. The body loop contains all iterations that access memory between the first and the last page boundary, with the head loop performing previous iterations starting from the lower bound of the original loop,

```

for(i=0; i<63; i++){
  Load A[i+1]
  Load B[i+1]
  ...
}
for(i=0; i<1280; i+=64){
  Load A[i+1]
  Load A[i+2]
  ...
  Load A[i+64]
  Load B[i+1]
  ...
}
(a) Unaligned

for(i=0; i<63; i++){
  Load A[i+1]
  Load B[i+1]
  ...
}
for(i=63; i<1279; i+=64){
  Load A[i+1]
  Load A[i+2]
  ...
  Load A[i+64]
  Load B[i+1]
  ...
}
for(i=1279; i<1280; i++){
  Load A[i+1]
  Load B[i+1]
  ...
}
(b) Aligned

```

**Figure 3: Alignment by Iteration Space Splitting**

and the tail loop computing subsequent iterations up to the upper bound of the original loop.

Figure 3(a) shows an example of an unrolled loop with misaligned memory references. Assuming that array  $A$  is aligned to a memory page, the memory accesses for one iteration of the unrolled loop span a page boundary. In (b), the iteration space of the original loop is split so that the memory accesses in the body loop start and end at page boundaries. The lower bound of the body loop and the lower bound of the tail loop are computed from the array subscript expressions and the loop bounds as follows. The earliest iteration where the most array references are aligned on a page boundary is used as the lower bound of the body loop. Let  $a$  be a representative reference to be aligned,  $l$  the loop index variable for the selected loop, and  $lb$  and  $ub$  the lower and upper bounds for  $l$ . To derive the loop bounds for the copies of the selected loop resulting from iteration space splitting, we begin with the starting address,  $addr$ , of the references when  $l = lb$ , where  $addr = aligned + offset$ . Here,  $aligned$  refers to the largest multiple of the page size less than  $addr$  and  $offset$  is the offset of  $addr$  within a page.

Assuming the stride of  $a$  is 1, the lower bounds of the body loop (*split1*) and the tail loop (*split2*) are computed by the following equations where  $P$  is the memory page size and  $T(a)$  is the type size of  $a$ .

$$split1 = lb + \frac{P}{T(a)} - \frac{offset \bmod P}{T(a)}$$

$$split2 = ub - (ub - split1) \bmod \frac{P}{T(a)}$$

The head loop is not needed if the reference is aligned, as is the case when  $offset \bmod P = 0$ . If  $lb$  is constant, *split1* and *split2* can be computed at compile time. Otherwise, they are computed at run time.

If the selected reference has non-unit stride, the solution is much more complex. In this case, we build a modular linear equation and choose the smallest solution [5].

**Reordering Memory Accesses** Finally, the reordering step hoists loads to the top of the loop body and sinks stores to the bottom. While being hoisted / sunk, the loads / stores to a same array are grouped together and sorted by their

```

for(i=32; i<N; i+=64){
  load A[i + 0] (RMA)
  load A[i + 32] (RMA)
  load A[i + 8] (RMA)
  load A[i + 40] (RMA)
  load A[i + 16] (RMA)
  load A[i + 48] (RMA)
  load A[i + 24] (RMA)
  load A[i + 56] (RMA)
  ...
}
(a) Unsorted

for(i=32; i<N; i+=64){
  load A[i + 0] (RMA)
  load A[i + 8]
  load A[i + 16]
  load A[i + 24]
  load A[i + 32] (RMA)
  load A[i + 40]
  load A[i + 48]
  load A[i + 56]
  ...
}
(b) Sorted

```

Figure 4: Sorting Offset Addresses

Parameters	Value	Unit
Random-mode latency	12	Cycles
Page-mode latency	4	Cycles
Page size	256	Bytes

Table 2: Simulation Parameters

offset addresses. When there are unaligned array references even after aligning the loop, sorting the offset addresses can reduce the number of random-mode accesses. Figure 4 shows an example where the page size includes 64 elements of array A. All eight memory accesses are in random mode before sorting. After sorting the offset addresses, only two random-mode accesses remain.

```

for(...){
  load A (RMA)
  load B (RMA)
  ...
  Computation
  ...
  store A (RMA)
  store B (RMA)
}
(a) Before

for(...){
  load A
  load B (RMA)
  ...
  Computation
  ...
  store B
  store A (RMA)
}
(b) After

```

Figure 5: Grouping loads and stores

This step also groups loads and stores to the same array when possible, to exploit page mode among them. There can be page-mode accesses between loads and stores if the last load and the first store access the same page, and there are no intervening memory accesses between them. The same is true between the last store of an iteration of the innermost loop and the first load of the next iteration. Using this technique, at most 2 random-mode accesses per iteration can be eliminated. Figure 5 (a) shows an example where two array objects are read and written. Assuming all loads and stores to the same array objects access the same memory page, the loop in (a) results in four random-mode accesses whereas (b) has only two random-mode accesses per iteration.

## 4. EXPERIMENTS

Although our algorithm is applicable to general embedded-DRAM systems, for the experiments presented in this paper we used a compiler framework that we have built for DIVA, as previously described. The DIVA PIM device has a 256-bit datapath for executing superword operations in parallel. In addition to a conventional scalar register file, the DIVA PIM processor has 32 256-bit registers (each of which can be

Name	Description	Input Size
VMM	Vector-matrix multiply	64 elements
MMM	Matrix-matrix multiply	64 elements
YUV	RGB to YUV conversion	32K elements
FIR	Finite impulse response filter	256 filter, 1K signal

Table 3: Benchmark programs

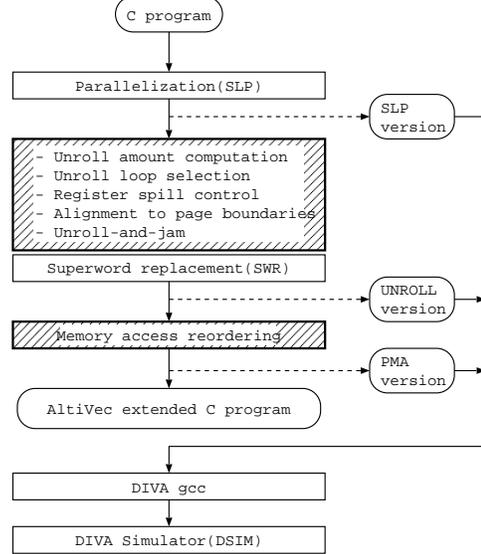


Figure 6: Experimental Flow

treated as eight 32-bit operands, sixteen 16-bit operands or 32 8-bit operands). Thus, for data allocated to superword registers, width  $W$  from Section 3 is 256, and for scalar registers  $W$  is 32. As the DIVA PIM devices contain no data cache, exploiting spatial locality in the memory pages can have significant impact on application performance.

A prototype of the DIVA PIM chip has been fabricated recently [6], but the complete DIVA system is not available for our experiments at the time of this writing. Therefore, we used a cycle-accurate DIVA simulator (DSIM) [6], which is modified from RSIM [20]. Table 2 shows the simulation parameters for the memory system which closely match those of the IBM Cu-11 embedded DRAM macro [14]. In general there can be multiple DRAM macros and multiple open pages in a single chip, but for our experiments we assume that only one memory page is open at any given time.

We implemented the bulk of the algorithm presented in the previous section, and integrated it into the Stanford SUIF compiler. In our current implementation, we have not implemented alignment to page boundaries or combining loads and stores for page mode accesses. However, these steps of the algorithm do not affect the results for the four benchmarks used. The input to the modified SUIF compiler is a C program, and the output is an AltiVec-extended C program [18] which, in turn, is translated by a preliminary version of the DIVA gcc backend.

Table 3 shows the four kernels used to evaluate the effectiveness of the algorithm. The kernels represent data intensive applications in scientific and multimedia domains. Figure 6 shows the experimental flow. The main algorithm involves selecting unroll factors, performing unroll-and-jam and memory access reordering, and is represented by the

hashed rectangles in Figure 6.

As previously stated, this algorithm is implemented as part of a compiler that exploits superword-level parallelism and locality in the superword register file. Thus, the experimental methodology also includes optimizations to exploit superword-level parallelism (SLP). Further, we exploit spatial and temporal locality in the superword register file through a combination of unroll-and-jam and *superword replacement* (SWR). Superword replacement is applied after unroll-and-jam to replace unnecessary superword memory accesses with references to superword temporaries that will then be allocated to superword registers by a backend compiler [22]. In our previous work, we selected unroll factors for unroll-and-jam that maximize reuse in superword registers; here, we use the unroll factors determined by the algorithm in Section 3, which are likely to be larger than in our previous work. In some sense, the optimizations for page mode memory accesses are complementary to exploiting SLP and locality in superword registers, and the page mode optimizations are difficult to isolate in our compiler. In fact, because the SLP and SWR optimizations reduce the number of memory accesses, we will see less benefit from the page mode optimizations than if considered in isolation.

We use as our baseline the SLP version of the code with no unrolling beyond what is required to exploit parallelization of the innermost loop. The UNROLL version includes unroll-and-jam, where the loop selected by the algorithm in Section 3 is unrolled by the chosen amount, and inner loop bodies are fused together. As compared to the baseline version, this version isolates the benefits of unroll-and-jam and superword replacement in terms of reduced memory accesses and less loop overhead, as compared to the baseline version. The PMA version reflects the performance improvements due to memory access reordering, yielding the full benefit of the optimizations for page-mode accesses.

In these experiments, we used optimization level -O1 for the DIVA gcc backend rather than a higher level of optimization. This was required to avoid reordering of memory accesses in subsequent optimization passes, which occurs at higher levels of optimization. Since reordering commonly occurs in backend optimizations, we discuss the implications of combining the page-mode optimizations with other backend compiler techniques in the next section.

For all programs but YUV, the algorithm was able to unroll the selected loop by the unroll factor determined by Equation 1. For YUV, which references six distinct arrays, this unroll factor was too large and resulted in register spilling. The algorithm reduced the unroll amount by half and the register spilling was eliminated.

We first consider how the optimizations for exploiting page-mode memory accesses impact memory stall time. In Figure 8 shows the normalized execution times broken down into processor busy time and memory stall time, derived from simulation. The UNROLL version sees a significant reduction in both processor busy time (9% to 60%) and memory stall time (25% to 71%). The primary reason for this is that superword replacement has eliminated a large number of memory accesses, which not only reduces memory stall time, but also reduce processor busy time by eliminating address calculation and instruction issue associated with the eliminated memory accesses. Further, reduction in loop control overhead also reduces processor busy time. For all programs, the PMA version further reduces memory stall

```

for(i = 0; i < 64; i++)
  for(j = 0; j < 64; j++)
    for(k = 0; k < 64; k += 8){
      load C[i][j]
      load B[i][k]
      load A[j][k]
      ...
      store C[i][j]
    }

```

(b) VMM

```

for(i = 0; i < 64; i++)
  for(j = 0; j < 64; j += 8)
    for(k = 0; k < 64; k++){
      load C[i][j]
      load A[i][k]
      load B[k][j]
      ...
      store C[i][j]
    }

```

(a) MMM

Figure 7: SLP versions of VMM and MMM

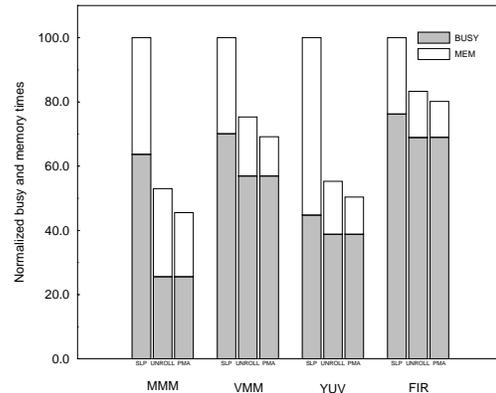


Figure 8: Normalized Execution Time

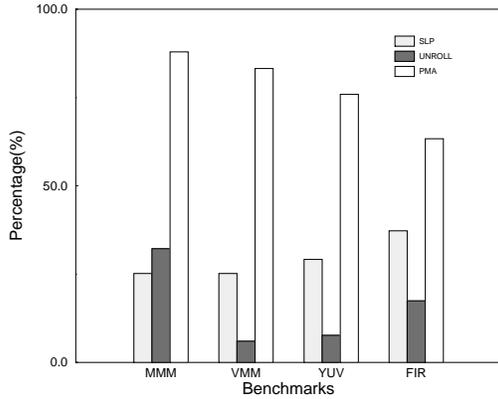


Figure 9: Percentage of Page-Mode Accesses

time by 21% to 33%. As compared to the UNROLL version, we have not eliminated any instructions, but rather have converted random-mode accesses to page-mode accesses.

Next we consider in Figure 9 the percentage of all memory accesses that are in page mode, with the remainder in random mode. The percentages of page-mode accesses ranges from 25% to 37% for the baseline version of the programs. We see a decrease in page-mode accesses as a percentage of memory accesses for most programs for the UNROLL version, ranging from 6% to 32%. This effect is because superword replacement has removed a large number of memory accesses, and the remainder tend to be in random mode. For example, in the VMM loop shown in Figure 7(a) after SLP, references to  $C[i][j]$  in the  $k$ -loop are loop-invariant after unrolling, and are usually removed, but were page-mode accesses in the SLP version due to the preceding store to the same location. In MMM, the page-mode percentage actually increases for the UNROLL version, as can be seen in Figure 7(b). References to  $A[i][k]$  are random-mode accesses, and are eliminated by superword replacement. For the PMA version, which reflects the same number of memory accesses as the UNROLL version, the percentages of page-mode accesses range from 63% to 87%.

These results show that our algorithm has been successful at increasing the percentage of page-mode accesses and reducing the memory stall time. We now see how the approach impacts the overall performance. Figure 10 shows the speedups for the SLP, UNROLL and PMA versions of Figure 6. Overall speedups as compared to the SLP baseline range from 1.25 to 2.19. Most of this speedup comes from the 1.19 to 1.89 improvement from unroll-and-jam and superword replacement, as can be seen from the UNROLL version. The speedup of the PMA version over the UNROLL version ranges from 1.04 to 1.16.

## 5. IMPLEMENTATION ISSUES

In this section, we consider in general terms how to incorporate this algorithm into current and future compilers. First, the compiler backend optimizations must be aware that page-mode optimizations are being performed. Otherwise, instruction reordering optimizations to increase instruction-level parallelism may undo the effect of the page-mode optimizations. A simple solution is to keep the relative order of memory operations intact when performing instruction reordering. There is an interesting tradeoff space that must

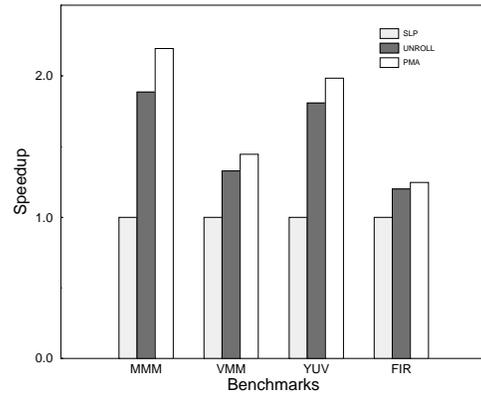


Figure 10: Speedup Breakdown

be considered, since page-mode optimizations which favor memory accesses to the same page may potentially lengthen the critical path to performing computations, where multiple operands from different pages may be required. This potential problem is mitigated if there are a large number of memory units that can operate in parallel, or if there are multiple memory pages from which data can be accessed rather than the single page used in our experiments.

A second issue is how to combine this approach with cache optimizations for devices that have on-chip data caches. In cache-based architectures, the page-mode optimizations are still applicable as long as the unrolled footprint for an object exceeds the cache line size. In such a case, the spatial locality within the memory page complements spatial locality within a cache line.

## 6. RELATED WORK

Previous research has identified the benefits of exploiting page-mode DRAM accesses [19, 17, 3, 21, 11]. Moyer modeled memory systems analytically and developed a compiler technique called *access ordering* that reorders memory accesses to better utilize the memory system [19]. McKee et al. described a Stream Memory Controller (SMC) whose access ordering circuitry attempts to maximize memory system performance based on the device characteristics [17]. Their compiler is used to detect streams but access ordering and issue is determined by the hardware. Chame et al. manually optimized an application for a PIM-based (embedded-DRAM) system [3] by applying loop unrolling and memory access reordering to increase the number of page-mode accesses.

Panda et al. have developed a series of techniques to exploit page-mode DRAM access in high-level synthesis [21]. Their techniques include scalar variable clustering, memory access reordering, hoisting and loop transformations. While their ASIC design was able to exploit page-mode memory access, they do not describe an algorithm for automatic code generation. Grun et al. have optimized a set of benchmarks to better utilize efficient memory access modes for their IP library based Design Space Exploration [11]. However, their focus was on accurate timing models of the hardware system description.

This paper is distinguished from previous research as the design and implementation of a compiler algorithm to ex-

plot page-mode automatically. Although the experiments are performed for a PIM-based system [12], this compiler framework is applicable to embedded-DRAM systems and can also be used as a preprocessor for high-level synthesis.

## 7. CONCLUSION

This paper presented a compiler algorithm for exploiting page-mode memory access in embedded-DRAM systems. Our compiler algorithm has been implemented in the Stanford SUIF compiler infrastructure and evaluated for four scientific and multimedia kernels. The speedups achieved by exploiting page-mode memory access alone range from 1.04 to 1.16 for four multimedia kernels, resulting in overall speedups ranging from 1.25 to 2.19 when combined with optimizations targeting superword-level parallelism and locality. These results show that there is a distinct benefit in exploiting page-mode memory access in embedded systems, where the DRAM access time dominates the memory latency seen by the processor. Furthermore, our results show that for embedded systems with support for superword-level parallelism [23, 1, 12], optimizations for exploiting the DRAM's page-mode accesses are complementary to optimizations for superword-level parallelism and superword-level locality.

## 8. REFERENCES

- [1] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ACM International Conference on Supercomputing (ICS'99)*, June 1999.
- [2] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.
- [3] Jacqueline Chame, Jaewook Shin, and Mary Hall. Compiler transformations for exploiting bandwidth in PIM-based systems. In *The 27th Annual International Symposium on Computer Architecture, Workshop on Solving the Memory Wall Problem*, June 11, 2000, Vancouver, British Columbia, Canada.
- [4] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [5] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 1990.
- [6] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steel, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 26–37, June 2002.
- [7] Karim Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.
- [8] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995.
- [9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
- [10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, October 1998.
- [11] Peter Grun, Nikil D. Dutt, and Alexandru Nicolau. Memory aware compilation through accurate timing extraction. In *Design Automation Conference*, pages 316–321, 2000.
- [12] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.
- [13] Mary W. Hall, Jennifer M. Anderson, Saman p. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.
- [14] IBM. *IBM Cu-11 embedded DRAM macro datasheet*, March 2002. [http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/4CBB96F927E2D6D287256B98004E1D98/\\$file/Cu11\\_Embedded\\_DRAM.pdf](http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/4CBB96F927E2D6D287256B98004E1D98/$file/Cu11_Embedded_DRAM.pdf).
- [15] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.
- [16] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC Canada, June 2000.
- [17] Sally A. McKee, William A. Wulf, James H. Aylor, Robert H. Klenke, Maximo H. Salinas, Sung I. Hong, and Dee A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, 2000.
- [18] Motorola. *AltiVec Technology Programming Interface Manual*, June 1999. <http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf>.
- [19] Steven A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, 1993.
- [20] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim reference manual. version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.
- [21] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. *IEEE Transactions on CAD*, February 1998.
- [22] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, September 2002.
- [23] Thomas Sterling. An introduction to the gilgamesh pim architecture. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2001.
- [24] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *ACM International Conference on Supercomputing*, Portland, OR, November 1993.
- [25] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.