Exploiting Off-Chip Memory Access Modes in High-Level Synthesis *

Preeti Ranjan Panda Nikil D. Dutt Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

Abstract

Memory-intensive behaviors often contain large arrays that are synthesized into off-chip memories. With the increasing gap between on-chip and off-chip memory access delays, it is imperative to exploit the efficient access mode features of modern-day memories (e.g., page-mode DRAMs) in order to alleviate the memory bandwidth bottleneck. Our work addresses this issue by: (a) modeling realistic off-chip memory access modes for High-Level Synthesis (HLS), (b) presenting algorithms to infer applicability of HLS with these memory access modes, and (c) transforming input behavior to provide further memory access optimizations during HLS. We demonstrate the utility of our approach using a suite of memory-intensive benchmarks with a realistic DRAM library module. Experimental results show a significant performance improvement (more than 40%) as a result of our optimization techniques.

1 Introduction

As library modules used in ASIC designs increase in complexity, existing high-level synthesis strategies will need to be modified to accommodate various aspects of complex interface protocols. One such complex module that has universal use in a wide variety of applications, is memory. Arrays in behavioral specifications are typically assigned to memories during synthesis. If these arrays are small enough, they may be mapped into on-chip RAMs. However, many applications involve large arrays, which need to be stored in off-chip memories, such as DRAM. Consequently, it is essential to employ a reasonably accurate model for memory operations during synthesis. Modern memories have efficient access modes (such as *page mode* and *read-modify-write*), that are known to improve the access bandwidth[6]. Since the newer generation of memories (Extended Data Out DRAMs, Synchronous DRAMs, etc.) all incorporate these access modes, it is important to develop a methodology for incorporating realistic memory library interface protocols into high-level synthesis, so that scheduling, allocation and binding algorithms can exploit these features to generate faster and more efficient designs.

Several
memory-related issues, such as memory *allocation*[2, 12], *packing*[9, 8], *estimation*[3], and *selection*[1] have been addressed in previous research on high-level synthesis. Most of them target register files (also referred in literature as *foreground memory*) and on-chip RAMs and ROMs (sometimes called *background memory*) that share space on the same chip as an ASIC. The problem of clustering of behavioral variables into single- and multi-ported register files has been addressed in [17, 10, 16].

In [11], a scheduling algorithm using *behavioral templates* is described. A behavioral template represents a complex operation in a Control Data Flow Graph (CDFG) by enclosing several individual CDFG nodes, and fixing their relative schedule. Templates allow a better representation of memory access operations than single multicycled CDFG nodes, and lead to better schedules. However, since different memory operations are treated independently, typical features of realistic memory modules (e.g., page mode for DRAM) cannot be exploited, because the scheduler has to assume the worst-case delay for every memory operation, resulting in relatively longer schedules.

In this paper, we make the following contributions towards incorporation of off-chip memory accesses in high-level synthesis:

1. We present models for the well-known operation modes of off-chip memories (e.g., various DRAMs), which can be effectively incorporated into HLS tools.
2. We present algorithms for inferring the applicability of the memory access modes to memory references in the input behavior.
3. We outline techniques for transforming the Control-Data Flow Graph (CDFG) for the input behavior to incorporate the memory access modes, so as to obtain an efficient schedule.

*This work was partially supported by grants from ARPA (MDA904-96-C-1472), NSF (MIP-9708067) and ONR (N00014-93-1-1348).

0-89791-993-9/97 $10.00 © 1997 IEEE

2 Memory Access Optimizations in DRAMs

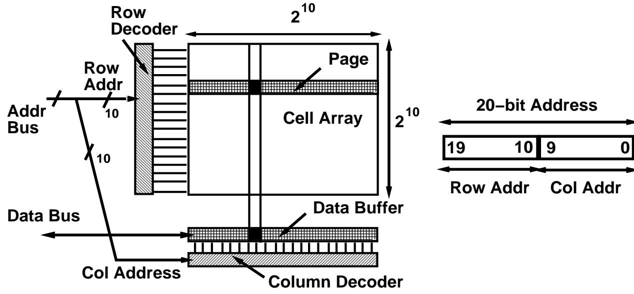


Figure 1. Typical DRAM Organization

Figure 1 shows a simplified view of the typical organization of a DRAM with 2^{20} words (= 1 M words), with the core storage array consisting of a $2^{10} \times 2^{10}$ square. A 20-bit logical address for accessing a data word in the DRAM is split into a 10-bit *Row Address* – consisting of the most significant 10-bits (bits 19 . . . 10); and a 10-bit *Column Address* – consisting of the least significant 10 bits (bits 9 . . . 0). The Row Decoder selects one of 2^{10} rows (or DRAM *pages*) using the row address, and the *Column Decoder* uses the column address to select the addressed word from the selected page. Since the row and column addresses are used over disjoint time intervals, the address bus is time-multiplexed between row and column addresses in order to save pin count. The following feature, called *page mode access*, is an important organizational attribute that results in significant access time reduction: *The row-decoding step physically copies an entire page into a data buffer (Figure 1), anticipating spatial locality, i.e., expecting future references to be from the same page. If the next memory access is to a word in the same page, the row-decoding phase can be omitted, and the data fetched directly from the data buffer, leading to a significant performance gain.*

Modern DRAMs commonly utilize the following six memory access modes:

Read Mode – single word read, involving both row-decode and column-decode.

Write Mode – single word write, involving both row-decode and column-decode.

Read-Modify-Write (R-M-W) Mode – single word update, involving read from an address, followed by write *to the same address*. This mode involves one row-decode and column-decode stages each, and is faster than two separate Read and Write accesses.

Page Mode Read – successive reads to multiple words in the same page.

Page Mode Write – successive writes to multiple words in the same page.

Page Mode Read-Modify-Write – successive R-M-W updates to multiple words in the same page.

In order to motivate the need for explicitly modeling and exploiting these modes, we demonstrate the effect on design performance in HLS between scheduling with normal read operation, versus scheduling that exploits the page mode read of a DRAM. The sample library memory module used in this paper is the IBM11T1645LP Extended Data Out (EDO) DRAM, and the input behavior is the *FindAverage* routine in Figure 3(a), where the scalar variable *av* is mapped to an on-chip register, and the array $b[0 \ldots 3]$ is stored in off-chip memory.

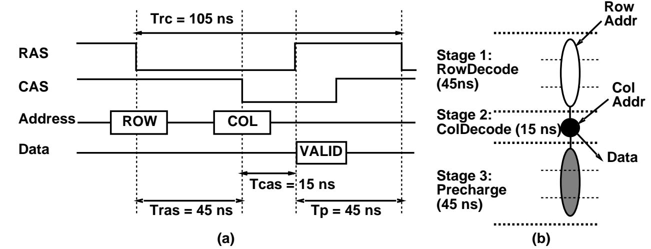


Figure 2. (a) Timing diagram for Memory Read cycle (b) Model for Memory Read operation

Figure 2(a) shows a simplified timing diagram of the *read cycle* of the $1M$ $\times$ 64-bit EDO DRAM. The Memory Read cycle is initiated by the falling edge of the RAS (Row Address Strobe) signal, at which time the row address is latched from the address bus. The column address is latched at the falling edge of CAS (Column Address Strobe) signal, which should occur at least $T_{ras} =$ 45 ns later. Following this, the data is available on the data bus after $T_{cas} =$ 15 ns. Finally, the RAS signal is held high for at least $T_p =$ 45 ns to allow for *bit-line precharge*, which is necessary before the next memory cycle can be initiated.

From the above timing characteristics, we can derive a CDFG node cluster for the memory read operation, which consists of 3 stages (Figure 2(b)): (1) row decode; (2) column decode; and (3) precharge. The row and column addresses are available at the first and second stages respectively, and the output data is available at the beginning of the third stage. Techniques for formally deriving the node clusters from interface timing diagrams have been studied in the interface synthesis works such as [4], and can be applied in this context.

Assuming a clock cycle of 15 ns, and a 1-cycle delay for the addition and shift operations, we derive the schedule shown in Figure 3(b) for the code in Figure 3(a), using the memory read model in Figure 2(b). Since the four accesses

to array b are treated as four independent memory reads, each of these incurs the entire read cycle delay of $T_{rc} = 105$ ns, i.e., 7 cycles, requiring a total of $7 \times 4 = 28$ cycles.

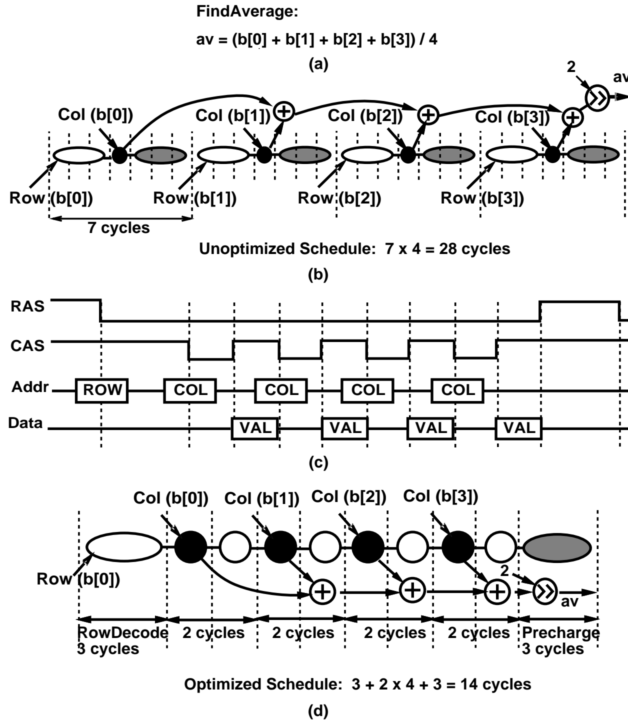


Figure 3. (a) Code for *FindAverage* (b) Treating the memory accesses as independent Reads (c) Timing diagram of *page mode read* cycle (d) Treating the memory accesses as one page mode read cycle

However, DRAM features such as page mode read can be efficiently exploited to generate a much tighter schedule for behaviors such as the *FindAverage* example, which access data in the same page, in succession. Figure 3(c) shows the timing diagram for the *page mode read* cycle, and Figure 3(d) shows the schedule for the *FindAverage* routine using the page mode read feature. Note that the page mode does not incur the long row decode and precharge times between successive accesses, thereby eliminating a significant amount of delay from the schedule. In this case, the column decode time is followed by a *minimum pulse width* duration for the CAS signal, which is also 15 ns in our example. Thus, the effective cycle time between successive memory accesses has been greatly reduced, resulting in an overall reduction of 50% in the total schedule length.

The key feature in the dramatic reduction of the schedule length in the example above is the recognition that the input behavior is characterized by memory access patterns that are amenable to the page mode feature, and the incorporation of this observation in the scheduling phase. In the following sections, we first provide synthesis models for the various access modes available in modern off-chip memories, and then describe a technique that incorporates the models into HLS by transforming the input behavior accordingly.

3 Representing Memory Accesses for HLS

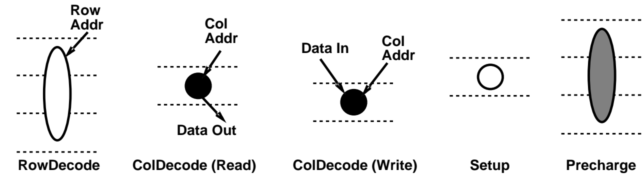


Figure 4. CDFG nodes used for memory operations

Instead of a single memory access node in the CDFG, we model memory operations corresponding to the different access modes using finer-grained memory access nodes (Figure 4). The **RowDecode** node is a multicycle operator that marks the beginning of any memory operation. The node has one input – the row address of the location, which should be ready before this node is scheduled. The **ColDecode (Read)** and **ColDecode (Write)** nodes form the second stage of the memory read and write operations. The **ColDecode (Read)** node has one input – the column address, and one output – the data read. The **ColDecode (Write)** node has two inputs – the column address, and the data to be written. The **Setup** node serves as a "delay" node in order to implement minimum delay constraints between successive stages of a memory operation. The **Precharge** node is a multicycle node that marks the last stage of a memory cycle. Physically, it signifies the restoration of the memory bit-lines to the initial state, so that the next operation can be initiated after the completion of this stage.

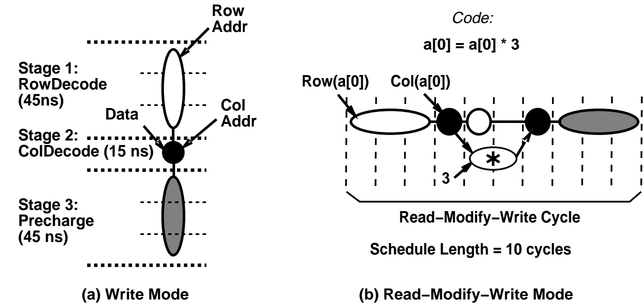


Figure 5. CDFG models of memory operations

For each memory access mode, we build composite memory access nodes in the CDFG, based on the access protocol for that mode.

Read Mode: This mode was described in Section 2.

Write Mode: The memory write operation has similar timing characteristics to the read operation. The CDFG

node cluster for the write cycle is shown in Figure 5(a). The ColDecode node used in the second stage is the **ColDecode (Write)** node identified in Figure 4.

Read-Modify-Write Mode: The Read-Modify-Write (R-M-W) mode is illustrated in Figure 5(b) with a simple behavioral statement: $a[0] = a[0] * 3$, which involves the reading and writing of memory address for $a[0]$. The schedule requires 10 cycles. Note that, an extra control step is introduced between the *setup* and *write* stages, because the '$*$'-operation requires 2 cycles. If the operation were to be modeled as separate read and write cycles, we would require: $2 \times 7 = 14$ cycles.

Page Mode Read: This mode was described in Section 2. The **Page Mode Write** and **Page Mode R-M-W** modes are constructed similarly.

4 Incorporation of Memory Models in HLS

The memory operations described in Section 3 were designed to exploit typical behavioral memory access patterns. This requires analysis of the CDFG to identify behavioral patterns that can be optimized by the various efficient memory access modes, and transformation of the CDFG to incorporate the optimizations. The actions that need to be performed include: (1) *clustering* of scalar variables; (2) *reordering* of memory accesses in the CDFG; (3) *hoisting* conditional memory operations; and (4) *loop transformations*.

4.1 Clustering of Scalars

Scalar variables are normally assigned to on-chip registers. However, if the number of such variables is large, it might be necessary to store these variables in memory. A related optimization problem that arises in this address assignment is that, consecutive accesses to two different scalar variables can be implemented as a single page mode operation if both are located in the same memory page. Suppose the off-chip memory has a page size of P words. The following problem needs to be solved: *Group the scalar variables into clusters of size P (to reside in the same memory page), such that the number of consecutive accesses to the same memory page is maximized*. The technique we use is similar to the solution of an analogous problem in the context of cache memory, where scalar variables are grouped into clusters of size L (where L is the size of a *cache line*), so as to minimize the number of cache misses[14].

At the end of this step, all variables are assigned a memory address. We assume, for convenience, that the first element of all arrays are aligned to a memory page boundary. Similarly, each row of a multi-dimensional array is padded so that all rows begin at a page boundary, unless the array is small enough to be accommodated in one page.

4.2 Reordering of Memory Accesses

The correct ordering of memory accesses is critical for exploiting efficient memory access modes such as R-M-W. For example, in the code: "$a[i] = b[i] + a[i]$", the sequence of accesses: *"Read $b[i]$ $\rightarrow$ Read $a[i]$ $\rightarrow$ Write $a[i]$"* allows the utilization of the R-M-W mode for the address $a[i]$, while the sequence *"Read $a[i]$ $\rightarrow$ Read $b[i]$ $\rightarrow$ Write $a[i]$"* does not allow the mode, because of the intervening *"Read $b[i]$"* operation.

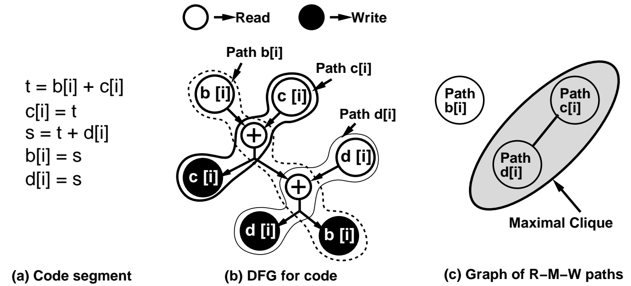


Figure 6. Ordering of memory operations determines possibility of exploiting R-M-W mode

Figure 6(a) shows an example basic block of code for which the corresponding DFG is shown in Figure 6(b). Figure 6(b) also shows three possible R-M-W paths, for addresses $b[i]$, $c[i]$, and $d[i]$. Note that it is not possible to implement the memory accesses to all the three locations as R-M-W operations. For example, if we implement the $c[i]$ path as an R-M-W operation, then the $b[i]$ path can no longer be implemented as R-M-W, because *"Read $b[i]$"* has to occur before *"Write $c[i]$"*, thus forcing the $b[i]$ path to be split into separate *"Read $b[i]$"* and *"Write $b[i]$"* operations.

In general, only one of a pair of intersecting paths can be implemented as R-M-W. In order to minimize the schedule length for a DFG, we need to maximize the number of R-M-W paths. The problem can be shown to be NP-complete by building a graph G, in which the nodes represent the DFG paths, and an edge exists between two nodes if the corresponding paths are non-intersecting (Figure 6(c)). Determining the maximal number of R-M-W paths in the DFG is now the problem of finding the *maximal clique* in G, which is known to be NP-complete. The greedy heuristic described in [15] can be used to solve this problem. However, since the number of such paths are very small in typical behaviors, an exhaustive solution can also be used.

4.3 Hoisting

Due to the time-multiplexing of the memory address bus between row and column addresses, a scheduling optimization is possible when two addresses in the same memory page are accessed from different paths of a conditional

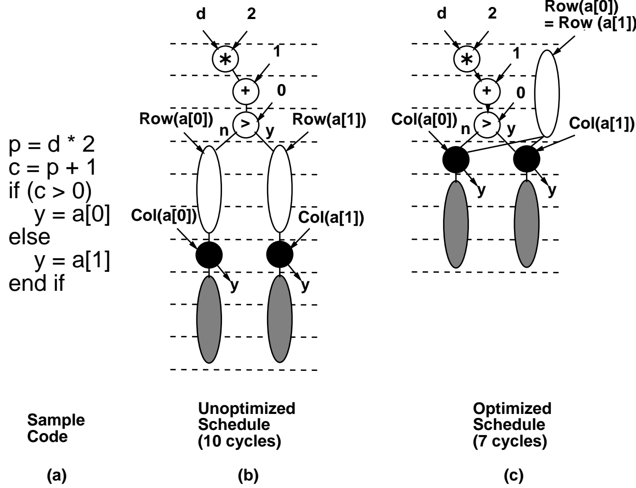


Figure 7. Hoisting Optimization

branch. Consider the behavior shown in Figure 7(a). Assume that variables p, c, d, and y are stored in on-chip registers, whereas array a is stored in off-chip memory. Either $a[0]$ or $a[1]$ is fetched from memory, depending on the result of the conditional evaluation ($c > 0$). A simple schedule, using the 3-stage read cycle model of Figure 2(b), and the assumption that $+$, $*$, and $>$ operations require 1 cycle, results in a schedule of length 10 cycles (Figure 7(b)). However, the knowledge that $a[0]$ and $a[1]$ reside in the same memory page (memory address assignments are statically computed) allows us to infer that both have the same row address, and hence, the read cycle could be initiated before the comparison operation. The schedule resulting from this optimization results in a length of only 7 cycles (Figure 7(c)). Output y is available after 7 cycles in the unoptimized schedule, but after only 4 cycles in the optimized schedule. We call this optimization – *hoisting* of the RowDecode stage. The *hoisting* optimization is also applicable for two elements x and y in the same page, accessed in different conditional paths C_1 and C_2 in the following cases: (1) x is written in C_1 and y is written in C_2; (2) x is read in C_1 and y is written in C_2; and (3) x is read or written in C_1, and there is no memory access in C_2[15].

4.4 Loop Transformations

In order to utilize the page mode operations, the CDFG for the behavior has to be transformed to reflect the page mode operation. Since most of the computation occurs in the innermost loops of nested loop structures, we concentrate on the memory accesses in the innermost loops.

Loops accessing single page per iteration

If a loop accesses locations from only a single memory page per iteration, e.g., there is only one read operation (*Read* $a[i]$) per iteration, the page mode operations can be applied directly by restructuring the loop so that it iterates over the array in blocks of P iterations (where P is the page size, in words), so that we have one page mode read for every P iterations. Figure 8(a) shows a section of the CDFG of an example loop with a single memory access ($a[i]$) in one iteration. We introduce an inner loop in which upto P elements from the same page are accessed. The transformed CDFG is shown in Figure 8(b). Note that the RowDecode and Precharge nodes enclose the inner CDFG loop, forming one complete page mode operation.

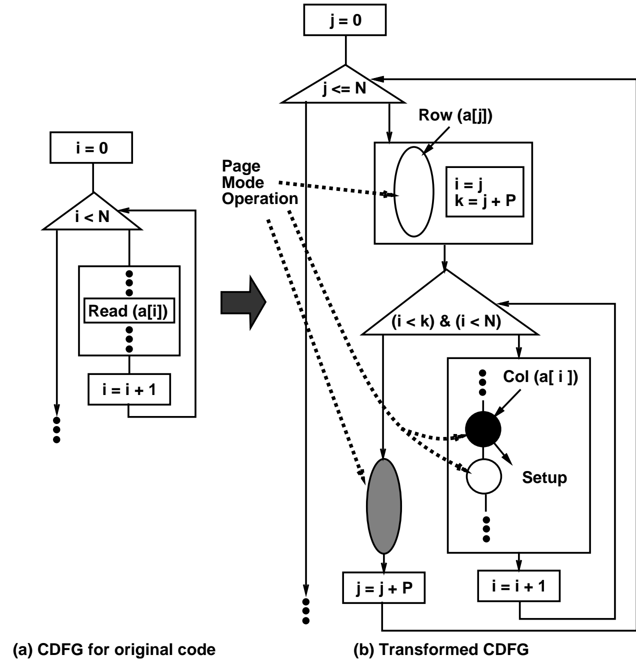


Figure 8. CDFG Transformation for page mode operation

Loops accessing multiple pages per iteration

If more than one array is accessed in one loop iteration, the transformation shown in Figure 8 cannot be directly applied, because different arrays usually lie in different memory pages. In such cases, we can use a well known transformation, *loop unrolling*[7], to create the opportunity for utilizing page mode operations. Figure 9(a) shows an example loop in which three different arrays, a, b, and c are accessed in an iteration. If the loop is unrolled once (Figure 9(b)), elements of the same array can be accessed in succession, leading to performance improvement resulting from page mode operation. In Figure 9, $r1, s1, t1$, etc. are registers into which memory elements are read in a *Read* operation, and from which memory elements are written in a *Write* operation.

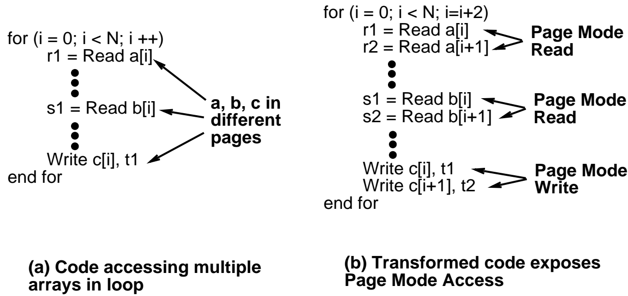


Figure 9. Loop unrolling helps exploit page mode operations

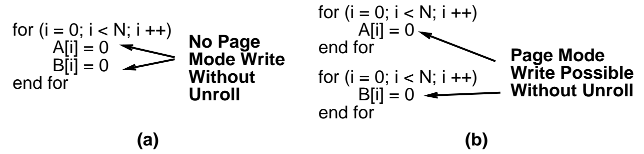


Figure 10. (a) Loop with disjoint subgraphs in body – there is no data dependence between the 2 statements (b) Split loop – page mode write can now be applied to the individual loops

Note that the unrolled loop in Figure 9(b) has a higher register requirement. Thus, the unrolling factor is constrained by the maximum number of on-chip registers available. Suppose we are given a register file of R registers. We first do an initial scheduling of the loop body (basic block) using *list scheduling*[7], to determine the number of registers, r, required for one iteration. Let m be the number of memory accesses in the loop body. Note that all memory accesses will not necessarily result in page mode accesses after unrolling. Let m' be the number of accesses that result in page mode accesses. If the loop is unrolled i times, we need $(m' \cdot i)$ registers to store the $(m' \cdot i)$ values. However, the remaining $(r - m')$ registers, which are used in the loop body to store temporary variables and non-page mode memory accesses, need not be duplicated, since they can be reused in the different iterations that constitute the unrolled loop. Thus, if the total number of registers allowed in the register file is R, we must have:

$$m' \cdot i + (r - m') \leq R \tag{1}$$

i.e., the loop unrolling factor, i, is given by:

$$i \leq \frac{R - r + m'}{m'} \tag{2}$$

or

$$i = \left\lfloor \frac{R - r + m'}{m'} \right\rfloor \tag{3}$$

Loops with disjoint subgraphs in body

A loop body is said to consist of disjoint subgraphs if the DFG representing the body (basic block) can be divided into more than one subgraph with no data dependence across their memory operations (i.e., no data dependence edges from one subgraph to the other). In such a case, each subgraph with at least one memory access in it, can be split into a different loop in order to better utilize page mode memory operation.

An example loop with disjoint subgraphs in its body is shown in Figure 10(a). The two statements in the loop body have no data dependence. We can split the loop into two loops, as shown in Figure 10(b), so that page mode write can now be applied to the individual loops without unrolling. Page mode operations without loop unrolling are preferable because unrolling increases the register requirements, and also limits the number of memory accesses in one page mode cycle, due to register file size constraints.

5 The CDFG Transformation Algorithm

Algorithm *TransformCDFG* in Figure 11 outlines the sequence of steps for transforming the CDFG of the behavior into an optimized form, so that a scheduling algorithm, such as list scheduling, can be invoked on the transformed CDFG. After performing the scalar variable clustering step, we perform the Reordering, Hoisting and Loop Splitting transformations in sequence, followed by restructuring of the resulting loops, if applicable. The complexity of the algorithm is that of the dominant loop restructuring (with multiple pages per iteration) step — $O(Bn^2)$, where n is the number of CDFG nodes and B is the number of loop nests in the behavior.

6 Experiments

We tested our proposed optimizations for utilizing efficient memory access features, on benchmark examples from the digital signal processing and scientific computing domains, all of which share the common characteristic that they process large data arrays. We present a summary of our experimental results in this section.

Column 1 of Table 1 shows the list of benchmark examples (taken from [13]) on which we performed our experiments. *Beam* (Beamformer) is a DSP application involving temporal alignment and summation of digitized signals from an N-element radar antenna array. *Dequant* is the de-quantization routine in the MPEG decoder application. *DHRC* (Differential Heat Release Computation) is an algorithm modeling the heat release in a combustion engine. *IDCT* (Inverse Discrete Cosine Transform), *LeafComp*, and *LeafPlus* are modules from the MPEG decoder application.

Algorithm *TransformCDFG*
Input: G – CDFG; R – Max. allowed
Register File Size; P – Memory Page Size
Output: Transformed CDFG
1.Cluster scalar variables into groups of size P
and assign memory addresses.
2.**for** each basic block B in G do
  Reorder to exploit R-M-W and page mode in B.
3.**for** each conditional node in G
  Perform *Hoist* transformation, if applicable.
4.**for** each innermost loop L in G
  Perform *Loop Splitting* transformation, if applicable.
5.**for** each loop L' in updated G
  if single page accessed in one iteration
    Perform *Loop Restructuring* for page mode
  else
    Perform *Loop Unroll* and *restructuring*

Figure 11. Algorithm for incorporating memory optimization transformations into the CDFG for scheduling

Madd and *MMult* are matrix addition and multiplication routines respectively. *Lowpass* is an image processing application that applies a low-pass filter to an image. *SOR* (Successive Over-Relaxation) is an algorithm used in evaluating partial differentiation equations. Column 2 shows the number of basic blocks in each benchmark.

Benchmark	B	Memory Modes				Optimizations			
		rmw	pr	pw	prmw	C	R	H	L
Beam	10	Y	Y	Y	N	N	Y	N	Y
Dequant	5	N	Y	Y	N	N	N	Y	Y
Dhrc	2	N	Y	N	N	N	N	N	N
Idct	13	N	Y	Y	N	N	N	N	Y
LeafComp	7	N	Y	Y	N	N	N	Y	Y
LeafPlus	5	N	Y	Y	N	N	N	Y	Y
Lowpass	4	Y	Y	N	N	N	Y	N	N
Madd	4	N	Y	Y	N	N	N	N	Y
MMult	6	N	Y	N	Y	N	N	N	Y
SOR	4	Y	Y	N	N	N	Y	N	N

Table 1. Memory optimizations applied to benchmarks

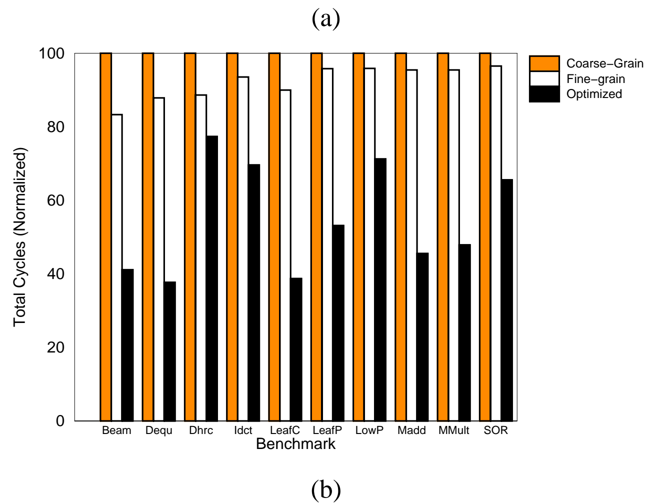
Table 1 shows the memory modes utilized by the memory accesses in the various benchmark examples, and the applicable optimizations. Columns 3, 4, 5, and 6 show (with letter 'Y') which examples had memory access patterns for which the Read-Modify-Write (rmw) mode, page mode read (pr), page mode write (pw), and page mode read-modify-write (prmw) respectively, were applied. Columns 7, 8, 9, and 10 show the CDFG transformation techniques – *Clustering*(C), *Reordering*(R), *Hoisting*(H), and *Loop Transformations*(L) – that were applied to each example. Note that the scalar clustering technique could not be applied to any of the examples, as the number of scalars in the example was too small, and could be stored in the register file we used.

We compared the execution time due the schedules generated by 3 techniques: (1) **Coarse-grain** – the traditional HLS approach, where memory access operation is treated as a multicycled operation; (2) **Fine-grain** – a more refined memory access model (e.g., the template strategy of [11]), with each memory access operation being treated as an independent 3-stage operation; and (3) **Optimized** – our proposed CDFG transformation-based approach, followed by the application of list scheduling.

We used the approximate timing characteristics of the IBM11T1645LP EDO DRAM memory chip, with a 15ns clock. We used the following operator delays: ALU – 1 cycle; multiplier – 2 cycles; divider – 4 cycles; RowDecode and Precharge operators – 3 cycles; and ColDecode and Setup operators – 1 cycle. We used a memory page size of 256 words and register file size of 16 words. We assumed that the number of ALUs, multipliers, and dividers is 1 each; and the register file has 2 read ports and 1 write port.

Benchmark	Cycle Count (Coarse-Grain)
Beamformer	1,251,844
Dequant	4,226
Dhrc	6,784
Idct	63,521,284
LeafComp	2,562
LeafPlus	3,074
LowPass	1,159,202
Madd	360,706
MMult	46,285,058
SOR	690,860

(a)



(b)

Figure 12. (a) Cycle count for Coarse-Grain (b) Summary of Results

Figure 12 summarizes the experimental results for the benchmark examples on which we tested our technique. Figure 12(a) shows the number of clock cycles required by

the coarse-grain technique for each benchmark. On the x-axis of Figure 12(b), we show the benchmark examples, and on the y-axis, we show the total execution times for the schedules generated by the three techniques: Coarse-grain, Fine-grain, and Optimized, normalized to the cycle count of Coarse-grain, which is taken as 100.

The Optimized technique results in the fastest schedules, as seen from Figure 12(b). On an average, Optimized achieves a performance improvement of 45.2% over Coarse-grain, and an improvement of 40.8% over the Fine-grain technique. The performance improvement in the case of Optimized is the consequence of efficient utilization of the memory features such as read-modify-write and page mode.

The performance improvement for the Optimized technique above could be at the expense of a slight increase in controller area due to increased number of control states as compared to the controllers required for the Coarse-grain and Fine-grain schedules when the page mode operation is used. We used the *misII* synthesis package[5] to optimize the control logic for each case, and studied the control overhead for our optimization technique. We observed that the controllers generated from the Optimized schedules have an average area only 14.9% larger than those generated from Coarse-grain, and 10.3% larger than those generated from Fine-grain, for the benchmark examples[15]. Given the significant improvement in performance with our optimizations, we believe the control overhead is justifiable.

An important property of dynamic RAM is that the internal data bits need to be refreshed at regular intervals, since each bit is implemented as a capacitor. The refresh intervals of typical DRAMs are fairly large — the IBM11T1645LP has a refresh interval of 128 ms. The time taken to perform a single *Refresh cycle* (which results in one whole DRAM page being refreshed) is comparable to the duration of any other access, such as *Read cycle*. As a post-processing step, the schedule generated from the input behavior is adjusted to incorporate the refresh cycles, ensuring that they do not overlap in time with any other memory operation, since the DRAM is unavailable for Reads and Writes during the refresh. This is not an overhead due to our approach because the refresh circuitry is needed for DRAMs, no matter what synthesis technique is used. The details are described in [15].

7 Conclusions

Off-chip memories, such as DRAMs, have several well-known features that permit efficient data access. Present-day synthesis tools and algorithms targeting on-chip memory storage treat different behavioral memory accesses independent of each other, thereby ignoring several optimization possibilities in memory accesses that arise in the context of off-chip memories. We presented synthesis models for various off-chip memory access modes, as well as a technique for analyzing a behavior to determine memory accesses that can be optimized by exploiting the available memory features. We transform the CDFG of the given behavior into an optimized form, incorporating the efficient memory access features. Our experiments, based on the timing characteristics of a commercial DRAM chip, have indicated an average performance improvement of more than 40%, as a result of our optimization techniques.

Our technique for incorporating the synthesis models for memories into HLS is independent of the actual HLS tasks of scheduling, etc., and can be easily incorporated as a pre-processing step into existing HLS design flows. The technique could also be utilized to optimize the interface with other types of modules with complex interfaces and timing characteristics, such as LAN controllers. Future research includes extending the optimization techniques to work in the presence of more complex array index expressions.

References

[1] S. Bakshi and D. Gajski, "A Memory Selection Algorithm for High-Performance Pipelines," Proceedings of EuroDAC, 1995.

[2] F. Balasa, et al., "Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems," Proc. ICCAD, 1994.

[3] F. Balasa, F. Catthoor, and H. D. Man, "Background Memory Area Estimation for Multidimensional Signal Processing Systems," IEEE Trans. on VLSI Systems, Vol. 3, No. 2, June 1995.

[4] P. Chou, R. Ortega, and G. Borriello, "Interface Co-Synthesis Techniques for Embedded Systems", Proc. ICCAD, Nov. 1995.

[5] R. K. Brayton, et al., "MIS: A multiple level logic optimization system," IEEE Trans. on CAD, vol. CAD-6, no. 6, Nov 1987.

[6] M. J. Flynn, "Computer architecture – pipelined and parallel processor design," Jones & Bartlett publishers, 1995.

[7] D. Gajski, et al., "High Level Synthesis: Introduction to Chip and System Design," Kluwer Academic Publishers, 1992.

[8] P. K. Jha and N. Dutt, "Library mapping for memories," European Design and Test Conference, March 1997.

[9] D. Karchmer and J. Rose, "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems," IEEE International Conference on Computer Aided Design, November, 1994.

[10] T. Kim and C. L. Liu, "Utilization of Multiport Memories in Data Path Synthesis," Design Automation Conference, 1993.

[11] T. Ly, et al., "Scheduling using Behavioral Templates," 32nd ACM/IEEE Design Automation Conference, June, 1995.

[12] P. E. R. Lippens, et al., "Allocation of Multiport Memories for Hierarchical Data Streams," Proc. ICCAD, November, 1993.

[13] P. R. Panda and N. D. Dutt, "1995 High Level Synthesis Design Repository," Intl. Symp. on System Synthesis, September 1995. (ftp://ftp.ics.uci.edu/pub/HLSynth95)

[14] P. R. Panda, et al., "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," ACM Transactions on DAES, Vol. 2, No. 4, October 1997.

[15] P. R. Panda, et al., "Exploiting Off-Chip Memory Access Modes in High-Level Synthesis," Tech. Rep. #97-32, U.C. Irvine, 1997.

[16] L. Ramachandran, D. Gajski, and V. Chaiyakul, "An algorithm for array variable clustering," Proc. ED&TC, 1994.

[17] L. Stok and J. A. G. Jess, "Foreground memory management in data path synthesis," International Journal of Circuit Theory and Applications, vol.20, no.3, pp. 235-55, 1992.