Memory Bank Customization and Assignment in Behavioral Synthesis

Preeti Ranjan Panda
Advanced Technology Group, Synopsys Inc.
700 East Middlefield Road, Mountain View, CA 94043, USA
E-mail: panda@synopsys.com

Abstract

With increasing design complexity and chip area, on-chip memory has become an important component whose integration needs to be addressed during system design. Modern embedded DRAM technology allows for large amounts of on-chip memory space. However, in order to utilize the available memory intelligently, the memory has to be appropriately customized for the specific application. We address the topic of incorporating the application-specific customization of memory bank configuration into behavioral synthesis. The strategy involves a partitioning of behavioral arrays into memory banks based on a cost function that estimates the performance implications. For a given candidate partition, we present a heuristic for determining the access sequence that minimizes page misses in a bank while respecting data dependences. The output of the exploration is a graph displaying the variation of delay and memory area with the bank configuration. Our experiments on several memory-intensive examples confirm that the exploration results can provide critical feedback to the designer about the optimal memory configuration for a given application.

1. Introduction

Effective utilization of chip area is an important issue in the design of application specific integrated circuits. With increasing design complexity and chip area, on-chip memory has become an important component whose integration needs to be addressed during system design [1]. Array variables in behavioral descriptions are usually mapped to memory locations when translated into hardware during the behavioral synthesis process. With the advent of embedded DRAM technology [15], the available on-chip memory can possibly satisfy the entire memory requirements of an application. However, in order to utilize the available memory intelligently, the memory has to be appropriately customized for the specific application. On-chip memory is amenable to this type of customization, since many memory parameters such as size, page width, bit width, and bank configuration are now controllable by the designer. This is in contrast to using off-the-shelf off-chip memory components in the design, where the designer or synthesis tool has no control over the internal organization of the memory. Knowledge about, and application-specific customization of the memory organization by a behavioral synthesis tool would be of immense help to the system designer by automatically evaluating the effect of various memory parameters on the area and performance of a given application, and suggesting the most promising memory configurations. In this paper, we address the issue of application-specific memory bank customization.

Memory banking is an organization strategy that helps increase the throughput of memory accesses by mapping different parts of the address space into different memory banks [4]. Some DSP processors, such as the Motorola 56000 have two on-chip memory banks, with a facility for parallel access from both banks in the same cycle. However, in all such architectures, the banking structure is determined in advance, and it is the responsibility of the designer to derive the most efficient memory mapping and access schedule for his application. Consequently, the performance of such applications is usually sub-optimal. A more attractive strategy is to tailor the number of banks and the assignment of variables to the banks to the specific application being synthesized, thereby increasing the possibility of optimal memory access performance.

Customization of on-chip memory has received the attention of researchers in recent years. Memory packing techniques for realizing the required memory for an application in terms of available components were presented in works such as [6, 5]. In [13], the authors present a technique for flow graph balancing, leading to lower overall memory bandwidth requirement for an application. In [9], the authors present a simulation-based technique for selecting a processor core and required memory and data caches for an application. Application-specific trade-offs between memory and CPU size for a given application were studied in [12].

Memory bank allocation of program variables in processors with dual-bank architectures were addressed in [14, 11]. While [14] addresses only scalar variables, [11] does not account for the access frequency of the variables. In [8], a method for allocating variables to dual-bank synchronous DRAMs was presented. All the above techniques work for dual-bank memories only. However, with the advent of embedded DRAM technology where DRAM can co-exist with other synthesized logic, several DRAM blocks can be treated as memory banks and exploited for parallel access, while simultaneously utilizing the advanced memory access features offered by DRAM technology, such as page mode access [10]. We present a strategy for exploring the effect of memory banking on the area and performance of an application during behavioral synthesis.

2. Memory Bank Customization

Memory bank customization can be illustrated by a simple example. Suppose we attempt to synthesize a loop of the form:

$$\text{for } i = 0 \text{ to } 1000$$
$$a[i] = a[i] + b[i] \times c[2i]$$

Since the arrays a, b, and c are, in general, too large to fit into a register file, they need to be stored in memory. Figure 1(a) shows one way of mapping the arrays into a single-bank memory, implemented in the form of embedded DRAM.

One important characteristic of DRAM architecture that should be accounted for during automatic memory mapping, is the presence of the *page buffer* (Figure 1(a)). The address presented to the memory is internally split into a row address (higher order bits) and a column address (lower order bits). The row address is used by a row decoder to select one full row (or *page* - we use row and page interchangeably in this paper for simplification of the discussion, although in reality, the two need not be the same), and copy it into the page buffer. The column address is decoded by a column decoder to select the offset of the addressed word within the page buffer . The architecture has the significant advantage that if subsequent memory accesses refer to words present in the page buffer (spatial locality), they can be directly accessed from the buffer, and the initial row decoding phase (as well as a final phase for precharging of the bit lines) can be omitted, thereby significantly speeding up the data access rate. This access mode is called *page mode* [10].

The page buffer has an important effect on the memory mapping strategy. In the example above, a naive approach of mapping the three arrays (a, b, and c) into a single-bank memory shown in Figure 1(a), is inefficient, since different pages corresponding to $a[i]$, $b[i]$, and $c[2i]$, are accessed in every iteration, overwriting the page buffer on every new access and destroying the locality of reference across different loop iterations. Work-arounds such as loop unrolling were suggested in [10], where, after unrolling an inner loop a few times, all references to an array can be read from the page buffer, exploiting a limited amount of spatial locality. For instance, in the for-loop above, elements $b[i]$, $b[i+1]$, and $b[i+2]$ can be accessed in sequence by unrolling the loop thrice.

However, loop unrolling is limited to a few iterations since the synthesized register file cannot be arbitrarily large. A more general strategy to exploit the DRAM page buffers in the presence of multiple array accesses in a loop iteration is to use banked memories. Figure 1(b) shows a three-bank memory architecture where the arrays a, b, and c are mapped into different banks. Since each bank is associated with an independent page buffer, there is no interference caused by different pages being accessed in the same loop iteration, and the effective access rate is much faster. For example, if the row decode, column decode, and precharge stages of a memory

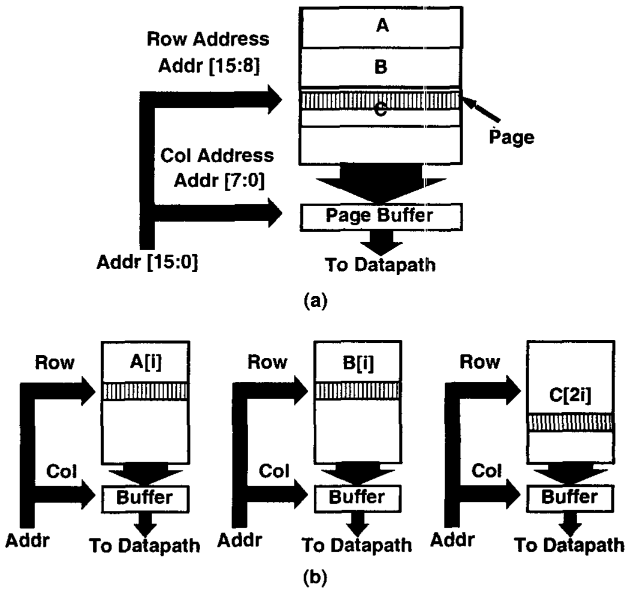

Figure 1. (a) Arrays mapped to single-bank memory (b) 3-bank memory architecture

access entail delays in the ratio 3 : 1 : 3 [10], the steady state throughput in the banked memory architecture is almost seven times that of the single-bank architecture.

Theoretically, it would seem that assigning a separate bank to each array would optimize the performance, but in practice, data dependences limit the amount of parallelism, resulting in no significant performance gain (and in fact, causing higher routing area penalty due to increased number of data and address buses) from an arbitrarily large number of banks. Further, in our example, interleaving the storage of $a[i]$ and $b[i]$ in the same bank is not a general solution since the arrays may be accessed in different ways in different loops. Note that there is no simple way to interleave $a[i]$ and $c[2i]$. A general memory customization strategy should consider the global effects of mapping arrays to banks over all loops.

In this paper, we present an algorithm for exploring the effect of application-specific memory bank customization on the area and performance of the synthesized design. The exploration algorithm analyzes several memory bank configurations, and for each, determines the assignment of variables to the banks as well as an estimate of the area and memory performance for the computed assignment. Since scalar variables can usually be stored in on-chip registers, we limit our analysis to arrays.

3. Exploration Algorithm

We wish to compute, in general, a graph showing the variation of area and delay with memory banking, so that a designer can select an appropriate point on the curve, which represents a bank configuration and the bank assignment of variables. The formulation can also be used to solve a variant of the problem stated as: given a maximum memory size constraint, determine a banking configuration and assignment that minimizes delay, which is a special case of the more general problem.

The basic strategy is to vary the number of banks in the architecture, and determine the best variable assignment and estimate the memory access performance for each configuration. The maximum number of banks M can be as large as the number of arrays, but for practical considerations, may be restricted to a constant, e.g., 8. We formulate the variable assignment to banks as a k-way partitioning problem of n-variables: determine a partition P_k of n variables into k groups such that a cost function *Delay*(P_k) is minimized.

We use as an overall strategy the k-way generalization of the Kernighan-Lin graph partitioning algorithm (also known as min-cut algorithm) [7, 2]. The algorithm is summarized below:

```
Algorithm Partition (G)
for k = 1 to M /* Do k-way partitioning */
  1. Generate initial partition P'.
  2. Generate n-move sequence into any of k partitions.
  3. Retain partition P_k with minimum Delay(P_k).
  4. Plot (k, Area (P_k)) and (k, Delay (P_k))
         on exploration graph
end for
end Algorithm
```

For each k, we start with an initial partition P' generated by a hierarchical clustering [2] of the set of arrays, where the decision to cluster two nodes (arrays) at each step is determined by the total number of times they are accessed in the same loop iteration. The pair for which this number is minimum is grouped into the same bank. Variables with non-overlapping lifetimes can be clustered in this step. The hierarchical procedure is illustrated in Figure 2, where each horizontal cut-line represents a partitioning of clusters into banks. We omit the details here due to lack of space.

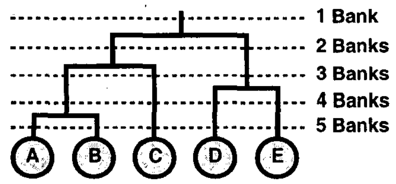

Figure 2. Cut lines assign clusters to banks

From the initial partition, we generate a sequence of n moves as follows: in each step of the sequence, we attempt to move any of the n nodes into any of k partitions – the move that minimizes the delay cost function is selected. Once a node is moved, it is not moved again in this step. We keep track of the partition P_k with minimum delay. The points (k, *Area* (P_k)) and (k, *Delay* (P_k)) are candidates on the exploration graph [1]. *Area* (P_k) is the total memory size required by the partitioning and bank assignment. For the same application, an increase in the number of memory banks could result in an increase or decrease in the required memory size since memory modules are designed to have sizes that are powers of two. For example, three 512B arrays require a 2K single-bank memory, but in the case of 2 banks, can utilize a 1K bank and a 512B bank, resulting in lower total area. Conversely, 3 arrays of 300B each can fit into a 1KB single bank memory, but if there are 3 banks, each requires a 512B bank, leading to greater total size. In the next section, we outline the procedure involved in computing the cost function *Delay*(P_k) used to evaluate a given partition P_k.

[1] In the original Kernighan-Lin algorithm, the steps 1-4 are enclosed in a 'forever' loop that terminates when there is no decrease in delay in a new partition. We omit this outer loop here in the interest of computational complexity of the exploration algorithm.

4. Cost Function Computation

The cost function *Delay*(P_k) used to evaluate the effectiveness of partition P_k is an estimate for the total length of the resulting schedule. We perform a simple list scheduling [2] of the inner loop DFGs to compute the estimate. However, at this stage, the scheduling cannot be performed because the memory access delays are still unknown. Normal accesses incur larger delays than page mode accesses (Section 2), and the type of access is known only after the sequence of memory accesses in each bank is determined. In the rest of this section, we present a technique for determining a good ordering of memory access for a given bank partitioning of arrays. Our objective is to find an ordering of memory accesses that minimizes the number of page misses, i.e., the number of times the accessed memory data is not already present in the data buffers. We assume that inter-basic block optimizations such as loop-invariant code motion, etc., have already been performed.

We start with a data flow graph for the basic block, and generate from it a Memory Dependence Graph (MDG), which is essentially a partial order of the memory access dependences within the basic block. An edge $u \rightarrow v$ in the MDG implies that access u should precede access v in the resulting ordering.

An example DFG and the corresponding MDG are shown in Figure 3(a) and (b). In essence, the MDG consists of the DFG with only the memory access nodes, with the dependences propagated through the non-memory nodes. Procedure *GenMDG* summarizes the procedure for generating the MDG.

```
Procedure GenMDG
Input – g: DFG
for all nodes i
  visited [i] = FALSE
  dep [i] = $\phi$
end for
for all nodes i
  if visited [i] == FALSE
    Visit (i)
end for
end Procedure
```

```
Procedure Visit
Input – i: node
for all parents nodes p of i
  Visit (p)
  if (isMemAccessNode(p))
    dep [i] = dep [i] ∪ {p}
  else
    dep [i] = dep [i] ∪ dep [p]
end for
visited [i] = TRUE
end Procedure
```

GenMDG is a depth-first procedure that maintains at each node the set (dep) of memory access nodes on which the cur-

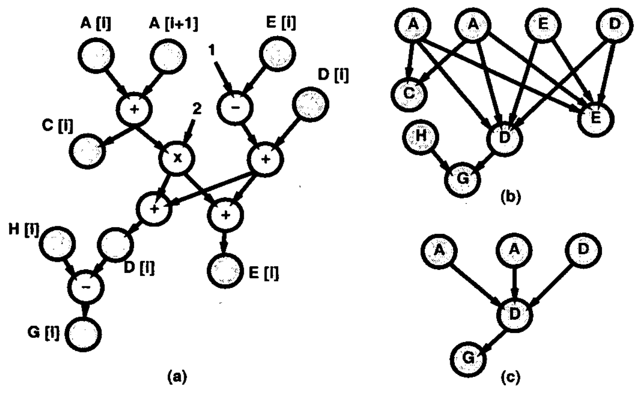

Figure 3. (a) Example DFG (b) Derived Memory Dependence Graph (MDG) (c) Partitioned MDG (PMDG) for bank 1 under the partition: [Bank 1: A,D,G], [Bank 2: C, F], [Bank 3: E]

rent node depends. Note that the depth-first search grows upwards from a given node in Figure 3(a). For a given node i, if the parent node p is a memory access, then we add p to the set dep[i]. On the other hand, if the parent is a non-memory node, then we add to dep[i] all members of dep[p], because, in this case, node p will not be present in the generated MDG, so its dependences are inherited by i.

The MDG forms the basis for the bank partitioning exploration. Once a partition P_k of arrays into k banks is generated, the corresponding MDGs for each basic block can be further partitioned into k sub-graphs. Figure 3(c) shows the partitioned MDG for bank 1 for a candidate partition: [Bank 1: A,D,G], [Bank 2: C, F], [Bank 3: E]. Note that a perfectly analogous procedure to *GenMDG* can be applied to the original MDG to generate the partitioned MDG for a specific bank, with the memory vs non-memory distinction in the treatment of nodes in *GenMDG* replaced by current bank vs. other banks.

The final step is to generate an ordering in the partitioned MDG (PMDG) that minimizes the number of page misses. Essentially, we look for a topological sort of the graph with the minimum number of page transitions. For the graph in Figure 3(c), the corresponding ordering is: $A \rightarrow A \rightarrow D \rightarrow D \rightarrow G$. This sequence of memory accesses minimizes the number of page misses in bank 1, while respecting the dependences in the PMDG. The problem of generating an optimal sequence for minimizing the number of page misses can be shown to be NP-hard. A degenerate case of this problem, formulated as the RMW-optimization problem, was shown in [10] to be equivalent to the clique partitioning problem [3].

Procedure *SchedulePMDG* below outlines a greedy method for generating a memory access sequence from a PMDG. The main idea is to schedule those accesses that lead to the longest sequence of accesses in the same page before scheduling an access to a different page. Set X consists of the set of unscheduled nodes in the same page (which can hence be scheduled in sequence, leading to page hits). Set Y is the set of all internal nodes, lying in the same page, that become schedulable after all nodes in X are scheduled. This is obtained by propagating (routine *Propagate* is omitted due to lack of space) the *schedulable* condition to all nodes lying in the same page whose parents are also schedulable. The cardinality of the set $X \cup Y (= |X \cup Y|)$ gives the length of the sequence of page mode operations. We select for scheduling the set with the largest $|X \cup Y|$. A schedule in this context means the addition of a precedence edge in the control flow graph to impose the relative ordering of memory accesses.

```
Procedure SchedulePMDG (g: PMDG)
G = PMDG
while ($G \neq \phi$)
  maxLength = 0
  for each schedulable node $s$
    Set $X = \{t | t$ is schedulable and Page($t$) = Page($s$)$\}$
    Set $Y$ = Propagate ($X$)
    if (maxLength < $|X \cup Y|$)
      maxLength = $|X \cup Y|$
      $Z = X \cup Y$
    end if
  end for
  Schedule $Z$
  $G = G - Z$
end while
end Procedure
```

The main loop in the min-cut-based exploration algorithm takes $O(n^2)$ time where n is the number of arrays, assuming M, the maximum number of banks, is a small constant such as 8. Procedure *GenMDG* requires $O(r^2)$ time, where r is the number of memory access nodes (assuming $r^2 > e$, the number of edges in the original DFG). *GenMDG* may add some new edges to the MDG, while at the same time, removing all non-memory nodes and their associated edges. In practice, we observe that the MDG is usually much smaller than the DFG because of the omission of all the computation-oriented nodes. Procedure *SchedulePMDG* could require upto $O(r^2)$ time because, in the worst case, $O(r)$ nodes might need to be looked up every time a new set X is evaluated for scheduling. Thus, the theoretical computational complexity of the exploration is $O(n^2 r^2)$. However in practice, the computation time can be improved in several ways. First, in step 2 of the main exploration algorithm, we can limit the sequence of moves to a constant instead of the general n. Also, if n is large, the initial hierarchical clustering of arrays can help reduce the effective number of nodes in the MDG. Finally, if r is small, *SchedulePMDG* can be replaced by an exhaustive search-based strategy.

5. Experiments

We used several memory access-intensive applications to demonstrate memory bank exploration. Table (1) gives the example names, the number of arrays and the number of memory and non-memory nodes in the DFGs of the inner loop bodies. IDCT and SOR are used in image processing. EQN_OF_STATE and 2D-Hydro are examples from the Livermore Loops suite of scientific computing benchmarks. Figure 4 shows the variation of the area and delay (normalized to 1.0) with the number of banks. The bank count ranges from 1 (all arrays in the same bank) to the total number of arrays (each array in a different bank).

There are several interesting observations, the most important of which is the shape of the delay curves. In all cases, the delay decreases with increasing bank count initially, but eventually, the curve flattens out, leading to little or no improvement. For example, in SOR, increasing the number of banks beyond 3 results in no performance improvement because of data dependences in the loop body. Such points on the curve represent potentially optimal configurations.

The area curves represent the total size of the memory modules required for a given configuration and the associated variable mapping. As observed in Section 3, the required size varies for different configurations - in IDCT, increasing banks leads to lower total area, while for certain configurations in SOR and 2D-Hydro, the converse is observed. For configurations with similar performance and area characteristics, the correct decision is to choose the lesser number of banks, since this involves lesser routing area due to fewer data and address buses in the design. These graphs can be of immense value to the designer who can make an intelligent memory configuration decision based on the exploration. In all cases, the exploration algorithm required less than 10 seconds on a SUN UltraSparc system. We observed that procedure *SchedulePMDG* generated the same memory access sequence as an exhaustive-search strategy for the examples we considered.

Example	#Arrays	#Mem. Acc	#Ops
IDCT	3	9	4
SOR	7	12	13
EQN_OF_STATE	4	9	16
2D-Hydro	9	50	40

Table 1. Profile of Examples

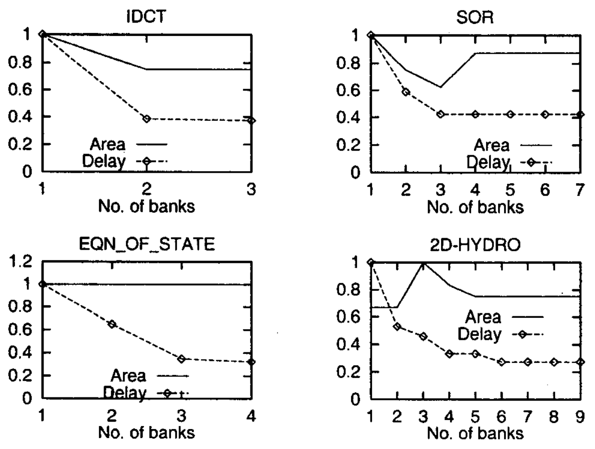

Figure 4. Memory Bank Exploration Results

6. Summary

We outlined a procedure for exploring the application-specific customization of memory banks during behavioral synthesis, based on the k-way min-cut graph partitioning algorithm. The evaluation of candidate architectures involves a fast heuristic for determining an assignment of array variables to memory banks with the objective of minimizing the number of page misses. In the future, we plan to employ a more realistic model for incorporating memory access delay and routing area (that varies with the bank structure) into the cost function. We also plan to investigate the integration of register allocation and banking when both register count and bank count are customizable.

References

[1] F. Balasa et al. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on VLSI Systems*, June 1995.

[2] D. Gajski et al. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[3] M. Garey and D. Johnson. *Computers and Intractibility - A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufman, 1994.

[5] P. K. Jha and N. Dutt. Library mapping for memories. In *ED&TC*, 1997.

[6] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *Proc. ICCAD*, 1994.

[7] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. Journal*, April 1970.

[8] A. Khare et al. High-level synthesis with synchronous and RAMBUS DRAMs. In *Proc. SASIMI'98*, February 1998.

[9] D. Kirovski et al. Application-driven synthesis of core-based systems. In *Proc. ICCAD*, 1997.

[10] P. R. Panda et al. Incorporating DRAM access modes into high-level synthesis. *IEEE Trans. on CAD*, February 1998.

[11] M. A. R. Saghir et al. Exploiting dual data-memory banks in digital signal processors. In *Proc. ASPLOS*, 1996.

[12] B. Shackleford et al. Memory-CPU size optimization for embedded system designs. In *DAC*, 1997.

[13] P. Slock et al. Fast and extensive system-level memory exploration for ATM applications. In *ISSS*, 1997.

[14] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs. In *Proc. ICCAD*, 1995.

[15] R. Wilson. Graphics IC vendors take a shot at embedded DRAM. *EE Times*, (938):41,57, January 27 1997.