Low Power Mapping of Behavioral Arrays to Multiple Memories

Preeti Ranjan Panda and Nikil D. Dutt
Department of Information and Computer Science
University of California, Irvine, CA 92717-3425, USA

Abstract

Large data arrays in behavioral specifications are usually mapped to off-chip memories during system synthesis. We address the problem of system power reduction through transition count minimization on the address bus during memory accesses, when mapping behavioral arrays to multiple memory modules drawn from a library. We formulate the problem as three logical-to-physical memory mapping subtasks, provide algorithms for each subtask, and present experiments that demonstrate the transition count reductions based on our approach. Our experiments show a transition count reduction by a factor of 1.5–6.7 over a straightforward mapping scheme.

1 Introduction and Related Work

The increasing demand for portable electronic equipment in recent times has led to extensive research in low power design techniques. Applications such as portable communications and multimedia drive the trend towards reduction in power dissipation in VLSI systems. Minimizing memory-related power forms an important task in system design, since most of these applications involve heavy memory traffic, which accounts for a significant fraction (upto 50 %) of the total system power [10]. Our focus in this paper is the reduction of signal transition count on the address bus during memory accesses, based on a static analysis of the behavior. This reduction leads to power minimization not only by way of reduced switched capacitance in the off-chip address bus drivers in the ASIC (Figure 1), but also in the form of reduced switching activity in the address buffers and decoders in the memory.

Compiler techniques dealing with reduction in the number of memory accesses during program execution [2], are directly applicable to the problem of power minimization, since reduction in the number of memory accesses directly implies a reduction in the number of switchings of the memory address bus, the data bus, and the core memory circuitry. [10] presents transformations on the initial specification into an optimized form for reducing the number of memory accesses. Related work on exploiting data encoding to reduce transition activity is presented in [8], where a data-encoding strategy is used to decrease the number of I/O transitions at the expense of a moderate increase in on-chip transitions, by analyzing data streams on I/O pins.

ISLPED 1996 Monterey CA USA
0-7803-3571-8/96/$5.00 ©1996

In [6], a technique for minimizing address bus transitions was presented that exploited regularity and spatial locality in the memory accesses in a behavioral description and determined the mapping of behavioral array references to physical locations in a single memory. The impact of different memory address mapping strategies (*row-major*, *column-major* [1], and *tile-based*) on power dissipation was investigated by comparing the bit transition counts on the address bus. The study showed that, to reduce off-chip transition count, we can perform some additional computation on-chip, if required, with negligible area, delay, and power overhead, because off-chip load capacitances are three orders of magnitudes larger than on-chip capacitances [3]. In this paper, we present a generalization of the array mapping techniques to the more realistic case where the target architecture consists of a library of multiple memories of different sizes.

2 Problem Definition and Approach

We wish to perform the assignment of behavioral arrays to addresses in physical memory, which consists of single-ported memory modules of different sizes. Figure 1 shows the model we assume for a typical memory-intensive system, synthesized into an ASIC (consisting of datapath and control blocks) and a group of memory modules. We have a pair of data and address buses connecting the ASIC to each memory. This configuration is common in performance-critical systems where the required data access rate exceeds the maximum access rate possible when only a single memory can be accessed at any instant of time (i.e., multiple memories connected to a single data bus).

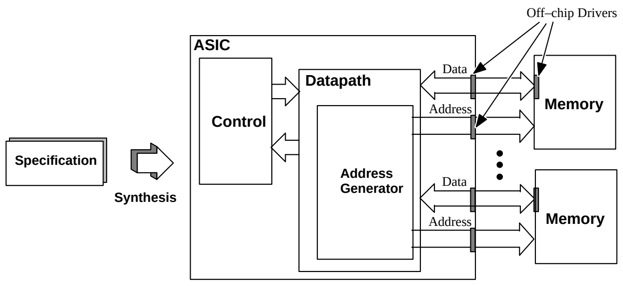


Figure 1: Synthesis model of a memory-intensive system

The approach we take is summarized in Figure 2. We first analyze the access patterns in the specification to determine the optimum partitioning of the arrays into multiple *logical array partitions*. E.g., in Figure 2, array A_1 is split into logical array partitions K_1 and K_2. We then regroup the logical array partitions into *logical memories* based on criteria such as the possibility of interleaving (Section 4). In Figure 2, logical array partitions K_1 and K_3 are merged into logical memory L_1. Finally, we map the logical memories into the available *physical memories*. The criterion here is minimization of the transition count overhead arising from mapping multiple logical memories to the same physical memory module. In Figure 2, logical memories L_1 and L_3 are assigned to physical memory P_1, whereas L_2 is assigned to P_2.

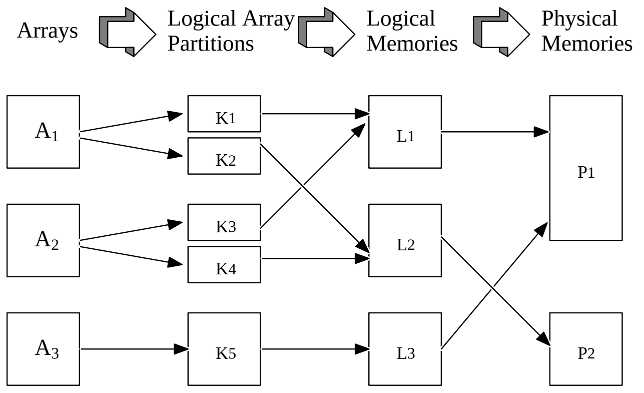


Figure 2: Mapping of Arrays to Multiple Physical Memories

3 Splitting into Logical Array Partitions

A procedure for organizing multi-dimensional arrays into *tiles*, based on an analysis of the access patterns in the behavior is described in [6]. Regularity in behavioral access patterns, which is a common feature in most memory-intensive applications, especially those in the DSP/Image processing domain, allows us to extract the dimensions of the tile for arrays accessed in loops. Figure 3 shows an array u, of dimensions 6×4, organized into four tiles. The tiles could have been derived, for instance, from an access pattern, in which the inner loop iteration of the behavior accesses elements (of u) from TILE1, followed by TILE2, TILE3, and TILE4. If array u is mapped to a single memory, spatial locality is improved during execution if the array is organized into tiles in memory (*tile-based* mapping), i.e., instead of the traditional row-major and column-major storage, elements of TILE1 are stored in consecutive locations, followed by those of TILE2, TILE3, and TILE4. In the multiple memories scenario, for each array, we use the tile thus derived as the starting point and split it into as many logical array partitions as number of array elements in the tile.

Figure 3 illustrates the splitting of array u whose tile contour has already been established, into multiple array partitions. We have six array partitions corresponding to each element of the tile. There are as many elements in each array partition, as the number of tiles in array u. The rationale for this division is that if each array partition is mapped to a different logical memory, this ensures optimality in terms of bit-transitions on the address bus. E.g., if $u[0][0]$ (in TILE1) is accessed in one loop iteration, we have $u[0][2]$, $u[3][0]$, and $u[3][2]$ being accessed in subsequent iterations (because the access pattern remains constant). Note that the partitioning in Figure 3 results in all these four elements being mapped into the same logical partition, thereby ensuring that consecutive elements of the partition are accessed in each iteration. If the memory address is converted into Gray code, the address bus for each memory would have just one bit transitioning between consecutive iterations of the inner loop of the specification.

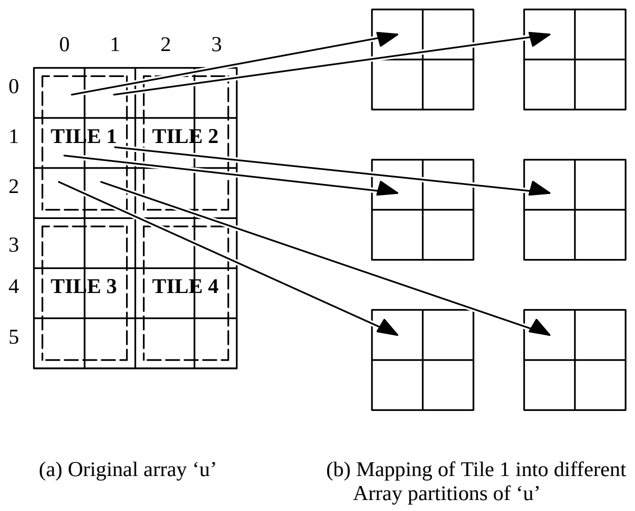

0
1
2
3
0
1
2
3
4
5
TILE 1
TILE 2
TILE 3
TILE 4
(a) Original array 'u'
(b) Mapping of Tile 1 into different Array partitions of 'u'

Figure 3: Partitioning of an array into multiple array partitions

4 Merging Logical Array Partitions

After partitioning the arrays in the specification to multiple logical array partitions, the next task is to group several array partitions into larger logical memories. This is because it might be prohibitively expensive to have a separate physical memory for each array partition. We consider the following properties while merging the array partitions:

Interleaving – It may be possible to interleave [1] multiple arrays in the same physical memory without suffering any penalty in bit transitions.

Independence – If two arrays are not accessed in the same loop, they can be stored in the same logical memory.

Non-overlapping Lifetimes – If two arrays have non-overlapping lifetimes they can be stored in the same memory locations, since the same memory space can be reused for the two arrays, i.e., we need only one logical array partition for the two arrays.

Based on the above properties, we construct a *compatibility graph* G in which each vertex represents an array partition, and the presence of an edge indicates that the two partitions can be placed in the same physical memory with no transition count overhead (i.e., they

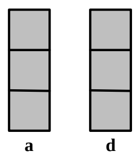
[1]When multiple arrays accessed in the inner loop have similar access patterns, we can *interleave* their storage to maintain spatial locality during accesses. Interleaving two arrays a and b means storing $a[i][j]$ and $b[i][j]$ in consecutive logical memory locations, followed by $a[i][j+1]$, $b[i][j+1]$, etc. (assuming row-major mapping).

are *compatible*). If two arrays are either independent, or can be interleaved, then we create an edge between the corresponding vertices.

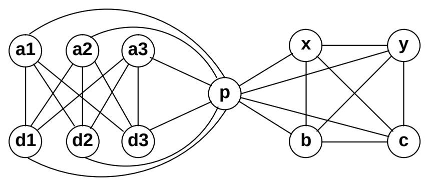
After constructing the compatibility graph G, we apply a *clique partitioning* algorithm to divide the graphs into sub-graphs of array partitions. A clique is a fully connected subgraph of the original graph G. The significance of a clique is that all partitions in the clique can be placed in the same physical memory. An exact solution of the clique partitioning is known to be NP-complete, so we employ an existing approximation algorithm [9] for this purpose. Each subgraph resulting from the clique partitioning corresponds to a logical memory.

```
for i in 2 to MAX–1 do
  for j in 1 to MAX do
    x[i][j] = a[i–1][j] * b[i][j] + a[i+1][j] * c[i][j] + a[i][j]
    y[i][j] = d[i–1][j] * b[i][j] + d[i+1][j] *c[i][j] + d[i][j]
for i in 2 to MAX–1 do
  for j in 1 to MAX do
    p[i][j] = p[i–1][j] + p[i+1][j]
```

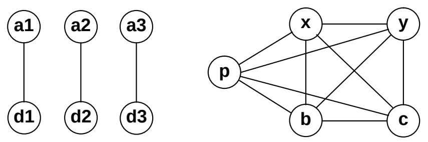
(a) Behavior with arrays a, b, c, d, x and y



(b) Tiles for ‘a’ and ‘d’



(c) Graph showing compatible array partitions. a1, a2, and a3 are partitions of array ‘a’. Partitions of array ‘d’ are d1, d2 and d3



(d) A possible partition into cliques – each clique represents a logical memory

Figure 4: Deriving logical memories from behavior

Figure 4(a) shows an example behavior and Figure 4(c) shows the corresponding graph for the array partitions. The tile corresponding to arrays a and d have three elements each, since 3 elements of each array are accessed in the inner loop (Fig. 4(b)). This leads to three array partitions for the two arrays. The array p has edges to all other array partitions, since it is accessed in a different loop, thus the independence property allows it to be placed in the same logical memory with any of the other array partitions. One possible partitioning of this graph into cliques is shown in Figure 4(d). This leads to four logical memories, each consisting of clusters of array partitions.

5 Mapping into Physical Memory

The next step is to map the logical memories to available physical memory modules. There exists a trade-off between memory utilization and address bus transition count: mapping each logical memory into a separate physical memory might be prohibitively expensive, forcing multiple logical memories to be mapped into the same physical memory. However, this mapping leads to inefficiency in terms of transition count, because some components of two different cliques in the compatibility graph would be incompatible (otherwise they would be in the same larger clique). We introduce an additional user-supplied constraint, a *memory packing factor f*, which allows the user to control this trade-off. The constraint *f*, which represents the minimum fraction of each physical memory that needs to be filled, can, of course, be set to 1, indicating that all the physical memories should be full. On the other hand, if the value of *f* is relaxed to less than 1, then a better packing in terms of transition count might be achieved. If *f* is set to 0, then the resulting mapping is optimum with respect to transition count, but could be area-expensive, as memory space would be wasted.

The algorithm we use for mapping the logical memories into physical memoriesis a variant of the *Best-Fit Decreasing* heuristic for the *bin-packing* problem [4]. The general strategy we adopt is to consider the logical memories one at a time, largest first and assign it to a physical memory module based on its own size and the sizes of the available memory modules. We start with the largest logical memory (L), since this is a candidate that possibly accounts for large transition counts. If L is larger than all available physical memories, we implement it with (multiple copies of) the largest available physical memory module. If there is a remainder of lesser size, we add it to the list of logical memories and continue the mapping process. If there is at least one physical memory module of greater size than that of the logical memory under consideration, we map the logical memory into the largest physical memory module that satisfies the memory packing factor *f*. If the constraint *f* cannot be satisfied, the mapping is too expensive, and we need to compromise on transition counts by mapping more than one logical memory to the same physical memory. The details of the algorithm can be found in [7].

6 Experiments and Results

We conducted experiments for testing the efficacy of our approach on several benchmark examples taken from the Image Processing applications domain [5]. We used the *cumulative count of the bit transitions* on the memory address bus during the execution of the behavior as a power consumption metric. In [6], a heuristic was presented to select an appropriate array mapping scheme, and experiments were reported, indicating a transition count reduction of 27 – 63% over the simple row-major mapping used by most compilers and synthesis tools. The experiments we report in this section demonstrate a further significant reduction in the multiple memories generalization.

Experiment 1 - In our first experiment to determine the impact of multiple memories on transition count, we used a configuration of multiple memories in the examples, where there is one physical memory available for each logical memory. In other words, this is the best improvement possible over the single memory case, since the partitioning into logical memories represents the ideal mapping, according to our technique. We recall, from Section 5, that we try to avoid, as far as possible, mapping of multiple logical memories to the same physical memory.

Table 1 shows a comparison of transition counts for five examples, all involving 2-dimensional arrays of dimensions (MAX $\times$ MAX), of the best mapping in the

Example	Transition Counts		Red.	Avg.
	Single Memory	Multiple Memories	(1000)	Red.
Compress	6518370	999002	6.5 X	6.6 X
GSR	28589669	8897227	3.2 X	2.9 X
Laplace	19653293	3997735	4.9 X	4.4 X
Lowpass	13218670	4993734	2.6 X	2.7 X
SOR	20438794	5997502	3.4 X	3.2 X

Table 1: Transition Count Reduction due to Multiple Memories

case of a single physical memory, and multiple physical memories. Columns 2 and 3 show transition counts for the different examples, with the value of MAX as 1000. Column 4 shows that the number of times by which transition counts decreased as a result of using multiple memories. Column 5 shows the average reduction for all the different sizes of the arrays that were considered for each example (MAX was varied from 50 to 1000 in steps of 10). We observe that transition count could be reduced by a factor of between 2.7 and 6.6 times if multiple memories are considered.
Experiment 2 - Experiment 1 demonstrates the possible transition count reduction in the hypothetical case of physical memories of the same size as the logical memories being available. In practice, however, the designer may be constrained by a specific library of physical memory modules. In the second experiment, we used a specific library of memory modules of sizes 128 KBytes, 256 KBytes, 512 KBytes, 1 MByte and 2 MBytes, with the assumption that each array element in the arrays of the examples occupies one byte (data bus is 8 bits wide). Table 2 shows the transition counts obtained for the same five examples when the mapping is performed with the above library.

Example	Transition Counts		Red.
	Simple Mapping	Our Mapping	
Compress	6701381	999002	6.7 X
GSR	19877346	11010165	1.8 X
Laplace	12884093	3997735	3.2 X
Lowpass	14060806	4993734	2.8 X
SOR	14118103	9145590	1.5 X

Table 2: Transition Count Reduction for a Specific Library

In Table 2, column 2 shows the transition counts obtained for the case when MAX = 1000 using a simple algorithm that maps the arrays in the specification (in order of decreasing size) to the smallest memory that accommodates them. Column 3 shows the corresponding transition counts using our approach. Comparing column 3 in Table 1 and Table 2, we note that for three of the examples (*Compress, Laplace* and *Lowpass*), the transition count in Experiment 2 was the same as the best case (one physical memory for each logical memory) transition count in Experiment 1 (i.e., our algorithm performed the optimal mapping). The transition count reduction factors shown in column 4 indicate a significant reduction resulting from our approach, ranging from a factor of 1.5 to 6.7.

7 Conclusions

In this paper we formulated and solved the problem of mapping of arrays in behavioral specifications to memories, in order to achieve address bus transition count minimization during memory accesses, when the target architecture includes multiple physical memory modules of different sizes drawn from a library. We conducted experiments on several image processing benchmarks exhibiting various (regular) memory access patterns. Our experiments showed that the power dissipation during memory accesses, as measured through off-chip signal transition count on the memory address bus reduced by a factor of 1.5 – 6.7 over a straightforward mapping scheme [2]. The mapping strategy we have described, is valid for array references in loops of the form $a[i \pm k]$, where i is the index and k is a constant. The formulation and solution of the problem to handle multi-dimensional arrays remains exactly the same.

8 Acknowledgment

We thank SRC for partial support of this work under SRC contract 95-DJ-146. We also thank Prof. Tomas Lang, Prof. Dan Hirschberg and Prof. Daniel Gajski for their valuable comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] D. F. Bacon, S. L. Graham and Oliver J. Sharp, "Compiler Transformations for High-Performance Computing," ACM Computing Surveys, Vol. 26, No. 4, December 1994.

[3] H.B. Bakoglu, "Circuits, Interconnections, and Packaging for VLSI," Addison-Wesley, 1988.

[4] E. G. Coffman, Jr. and G. S. Lueker, "Probabilistic Analysis of Packing and Partitioning Algorithms," Wiley-Interscience, 1991.

[5] P. R. Panda and N. D. Dutt, "1995 High Level Synthesis Design Repository," *International Symposium on System Synthesis*, Cannes, September 1995.

[6] P. R. Panda and N. D. Dutt, "Reducing Address Bus Transitions for Low Power Memory Mapping," *European Design and Test Conference*, Paris, March 1996.

[7] P. R. Panda and N. D. Dutt, "Low Power Memory Mapping through Reduced Address Bus Activity," Technical Report #95-32, UC Irvine, November 1995.

[8] M. R. Stan and W. P. Burleson, "Bus-Invert Coding for low-power I/O," *IEEE Transactions on VLSI Systems*, March 1995.

[9] C. Tseng and D. P. Siewiorek, "Automated Synthesis of Datapaths in Digital Systems," *IEEE Transactions on Computer Aided Design*, July 1986.

[10] S. Wuytack, et. al., "Global communication and memory optimizing transformations for low power systems," *IEEE International Workshop on Low Power Design*, Napa, CA, April 1994.

[2] In [6], it was shown that the overhead incurred in the address generator for the mapping schemes, in terms of area, delay, and power is negligible.