Automatic Synthesis of Customized Local Memories for Multicluster Application Accelerators

Manjunath Kudlur, Kevin Fan, Michael Chu, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{kvman, fank, mchu, mahlke}@umich.edu

Abstract

Distributed local memories, or scratchpads, have been shown to effectively reduce cost and power consumption of application-specific accelerators while maintaining performance. The design of the local memory organization must take several factors into account, including the memory bandwidth and size requirements of the program and the distribution of program data among the memories. In addition, when register structures and function units in the accelerator are clustered, the effects of intercluster communication should be taken into account. This work proposes a technique to synthesize the local memory architecture of a clustered accelerator using a phase-ordered approach. First, the dataflow graph is pre-partitioned to define a performance-centric grouping of the operations. Second, memory synthesis is performed by combining multiple data structures into a set of physical memories that minimizes cost while maintaining a performance threshold. Finally, post-partitioning is performed to determine the final assignment of operations to clusters given the memory organization. Results show that customization reduces memory cost from 2% to 59% over a naïve scheme that utilizes one physical memory per program data structure. Further, pre-partitioning is shown to reduce the intercluster communication required to achieve a fixed performance.

1 Introduction

Many portable devices must be capable of performing computationally demanding processing of images, sound, video, or packet streams. The challenging performance requirements must be met under tight power and cost constraints. Application accelerators in the form of application-specific instruction processors (ASIPs) and application-specific integrated circuits (ASICs) have been used with great success to meet the challenging demands of these systems. The accelerators are used to execute critical parts of applications that would run too slowly if implemented in software on an embedded processor. A customized accelerator is designed such that its computation capabilities are highly specialized to meet the specific needs of an application.

The design of the data memory system is a critical issue for application accelerators. Many embedded applications are characterized by large volumes of data access interleaved with computation. High bandwidth and fast access are required to achieve the desired performance levels. However, these objectives cannot be achieved by using large, centralized, multi-ported memory structures. Such designs are far too costly and often ineffective due to their complex designs and large access latencies. Distributed local memories (or scratchpads) offer the possibility of achieving the desired data processing rates and access latencies at large cost and power savings. Data is distributed across a set of local memories that are placed near the computation elements that require the data. High bandwidth is achieved by accessing the memory structures in parallel. Low latency is achieved by grouping the memory structures with the function units that require the data and restricting communications to geographic proximities, thereby forming *clusters*.

The central goal of this work is to synthesize a clustered local memory organization to meet desired bandwidth, size, and access requirements of a particular application at minimal cost. Traditional methods cannot be used as they ignore scheduling constraints and non-uniform access latencies of clustered architectures. Several interrelated problems must be solved to synthesize a customized

clustered memory organization: the number, size, and porting of local memories; the location of the local memories within the datapath clusters; and the distribution of data across local memories. Our approach to address this complex problem is to break the problem into simpler sub-problems that are solved in a phase-ordered manner: pre-partitioning, memory synthesis, and partitioning. Initially, each data structure (array or scalar) is placed into its own virtual local memory that has no cluster access restrictions. A performance-centric dataflow graph partitioning phase is done to pre-partition operations, including the virtual memory accesses, to datapath clusters. Memory synthesis is performed on the prebound memory accesses, combining virtual local memories to form a set of physical memories for each cluster. Finally, a second partitioning phase assigns operations to clusters with a fixed local memory organization to create the final specification of the architecture for the ASIC (or assembly code for the ASIP).

There are a number of research efforts that have investigated application-specific memory synthesis. Estimation techniques to calculate the storage requirements from a program with array computations have been proposed [28, 29]. Compiler optimizations and transformation techniques can also be used to reduce the inherent memory requirements [4, 13]. The problem of mapping arrays to one or more memories in hardware implementations has been addressed extensively in prior work. Some high-level synthesis systems map arrays into a single monolithic memory [6, 7, 8, 14]. Others take the opposite approach of mapping each array to its own memory [26]. Mid-point solutions to this problem wherein arrays are heuristically combined into multiple memories under performance constraints have also been proposed [17, 19]. Techniques for mapping multiple arrays to memories while reducing power consumption are described in [2, 15].

Architectural exploration techniques for datapaths with hierarchical memory systems and processors with local memory have been investigated [5, 10, 16]. More recently, it has been shown that it is important to consider factors such as performance, communication costs, and the scheduling effects during memory synthesis. In [11], a methodology is proposed to partitioning operations along with associated data in critical loops to arrive at a jointly optimized distributed memory organization for cost and performance. In [24], joint memory allocation and scheduling (assignment) can take advantage of non-uniform access speeds among memory ports to greatly diversify the possible memory organizations explored. Memory organizations for more complex abstract data types and dynamically allocated memory are described in [23, 25]. Comprehensive storage exploration methodologies that address many of the issues described in these works have been developed in the DTSE and ATOMIUM projects [5, 12].

Our work derives its roots from [11] and [17]. This work extends these techniques to perform synthesis for a class of clustered datapath/memory accelerator architectures.

2 Background

Throughput-directed Accelerator Synthesis. The synthesis approach used in this paper is derived from the PICO-NPA (Program-In Chip-Out nonprogrammable accelerator) synthesis system [20, 21]. PICO-NPA automatically synthesizes hardware accelerators for functions expressed as loop nests written in C. An NPA consists of a special-purpose array of one or more synchronous datapath processors along with their controller and possible local memories.

The underlying NPA datapath processor is synthesized using a throughput-directed approach. The original loop nest is collapsed down into a single nest which is then modulo scheduled [18]. Modulo scheduling is a technique to schedule an innermost loop by overlapping successive iterations multiple times. The throughput is specified as the constant time interval between starting successive iterations of the loop, known as the initiation interval (II). The target II is limited by the recurrence-constrained lower bound (RecMII), or the longest dependence cycle in the loop. The datapath is synthesized directly from the modulo scheduled code. Function units (FUs) are allocated to implement all of the operations in the loop body subject to the II. Operands are materialized using synchronous register FIFOs to hold loop-varying values, static registers for the live-in values, and hardwired literals. Registers are interconnected to FUs for each edge in the dataflow graph. Note that when the II is larger than one, several operations can map to each of the FUs, thus several operands are routed into and out of each FU using multiplexors.

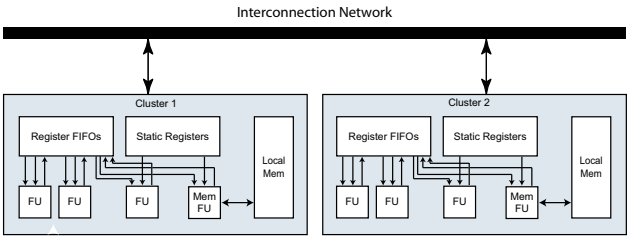

Figure 1. A multicluster NPA datapath processor organization.

Clustered Design. We extend the PICO datapath template from a flat design (or single cluster) to a distributed, multicluster datapath architecture as shown in Figure 1. Multicluster datapaths partition the design into several tightly coupled groups of FUs and register structures. Within a cluster, direct communication between any register/FU pair is performed in a single cycle. Intercluster communication is possible, but is performed through a low-bandwidth interconnection network. By decentralizing the datapath, the multicluster architecture reduces design complexity and decreases communication latency. All point-to-point communication is kept local by allowing it to occur only between operations in the same cluster. Explicit latency and bandwidth constraints are inserted into the schedule for intercluster communication. The resultant distributed datapath has both non-uniform connectivity and latency that must be accounted for during synthesis.

The benefits of distributed datapaths come at the expense of increased compiler complexity. The compiler must be cognizant of the cluster each operation is placed on, as placing producer/consumer operations on separate clusters will require data to be transferred with explicit intercluster move operations. Intercluster moves have a non-zero latency and can thus lengthen the schedule (e.g., increase II). However, if the latency of the intercluster moves can be kept off the longest recurrence paths, the intercluster moves will not seriously affect performance. Thus, a good partitioning of operations minimizes overall schedule length by simultaneously maximizing the number of operations executed in parallel while minimizing the affects of intercluster moves. Many different algorithms for effective partitioning of code into clusters have been studied in the past. The most well-known of these is the Bottom-Up Greedy (BUG) algorithm [9], which recurses depth-first along the dataflow graph, critical paths first. BUG uses estimates of the resource usage to greedily bind operations to clusters that minimize the estimated overall schedule length.

3 Virtual to Physical Mapping of Local Memories

The goal of this work is to produce lowest cost memory system that sustains a desired performance (e.g., the II) on a multicluster loop accelerator. The throughput-directed synthesis method starts with an initial configuration in which each data structure (array or scalar) is in its own virtual local memory. Memory synthesis consists of mapping multiple virtual memories into a set of physical memories that minimizes cost while sustaining the target II. The trivial mapping of each virtual memory realized as one physical memory is generally not cost effective. Cost savings can be achieved by combining several virtual memories into one physical memory. However, intelligent combining must be performed as many combinations are not cost effective or may violate the II constraints. The final assignment of virtual memories to physical memories determines the physical memory architecture of the machine, including the number and contents of each memory as well as the connectivity between memories and the rest of the datapath.

3.1 Issues

There are several issues which must be considered when allocating virtual memories to physical memories. For simplicity, the rest of this section considers arrays as the only data structures accessed by a loop nest.

Local memory cost. A primary objective is to minimize the cost of the physical memories required to realize a given set of arrays. Placing multiple program arrays into one memory can be

cost-effective as it saves on the overhead of the memory access hardware required for each memory. Examples of this hardware include the row decoders required to access a given memory address as well as the sense amps required to read out data. However, combining multiple program arrays into a single physical memory can also have negative effects on the memory cost. The size of the memory is larger; the width of the memory may increase as well, since it must be the maximum of the bitwidths of the contained arrays. In addition, the number of memory ports must be increased to maintain the desired II if the total number of accesses to program arrays in a given loop exceeds some threshold. Specifically, if there are acc accesses in a loop to arrays in a single physical local memory, $\lceil acc/II \rceil$ ports are required on that memory in order to sustain a throughput of II cycles.

Multicluster effects. In a multicluster datapath design, the placement of program arrays also has an effect on intercluster communication. It is desirable to spread operations across clusters in a balanced manner in order to distribute the computation. However, if array data is stored on a memory in cluster $C1$, and any operations which produce or consume that data are scheduled on cluster $C2$, explicit move instructions must be inserted to transfer values between the clusters. If these move operations occur on a recurrence path in the loop's dataflow graph, it may no longer be possible to achieve the desired II for the loop. Furthermore, even if move instructions do not occur on a critical recurrence path, there is a limited number of move instructions that can be executed by the machine in each cycle (known as the MAX_{ICM}). Thus no more than $(II \times MAX_{ICM})$ total intercluster moves can be scheduled in the loop.

Phase ordering. In determining the best allocation of local memories within a distributed datapath, a major concern is the proper phase ordering for making design decisions. Both the allocation of local memories as well as the partitioning of operations to clusters need to occur; however, they are highly intertwined decisions. By first deciding on the local memories that should exist in the datapath, limitations are made on the operation partitioning, as all loads for certain arrays must occur in specific clusters. Such limitations could have large ramifications on performance. Conversely, partitioning the operations first could limit the amount of cost savings that can occur from combining only arrays within the same cluster into local memories. Thus, it is difficult to separate the partitioning and synthesis phases.

3.2 Approach

The intertwined nature of synthesis and partitioning has many of the same characteristics of compiler scheduling and register allocation. While performing them jointly may make sense, the software complexity of the joint solution is too large. Thus, we adopt a strategy to enable each to consider the effects of the other, but still perform them separately by performing both pre- and post-partitioning. Each of the steps is described below:

1. **Pre-partition.** As mentioned in the previous section, local memory allocation decisions can affect performance on architectures with distributed datapaths. To take this into account, all operations in the loop body, including load/store operations are initially pre-partitioned into clusters assuming a single unified memory with infinite ports and uniform access latency on all clusters. This gives an optimistic measure of performance without memory constraints which the synthesis phase can use to determine performance degradation when mapping the virtual memories to physical memories.
2. **Synthesize memories.** Using the method described in Section 4, multiple virtual memories are intelligently combined into physical memories, and the physical memory organizations for each cluster are defined. Note that this may change the cluster assignments of load/store operations.
3. **Partition.** Finally, with the memory system fixed and the corresponding load and store operations bound to clusters, the remaining computation operations are partitioned and all operations are finally modulo scheduled.

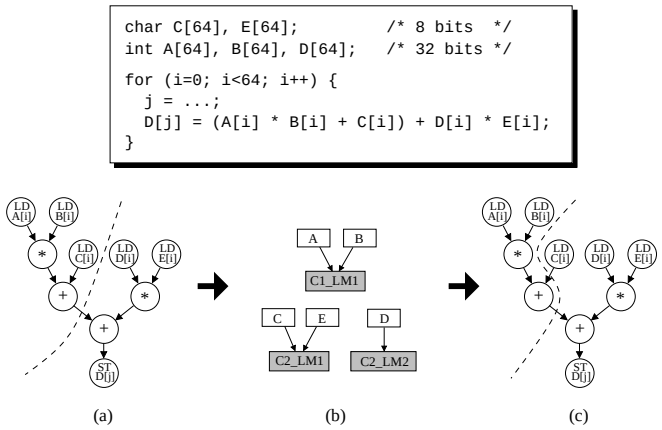

Figure 2. Synthesizing local memories for an example loop: (a) pre-partitioned dataflow graph, (b) synthesized local memory organization, (c) final partitioned dataflow graph.

3.3 Example

To demonstrate the local memory architecture design process, the source code of an example loop is shown at the top of Figure 2. Each array that is referenced by the loop occupies a virtual local memory whose bitwidth, size, and number of ports are determined by the contents of the array and the number of accesses to the array in the loop body. Note that breaking up of big arrays into sub-arrays and allocating arrays whose lifetimes do not overlap to the same memory are not addressed in this work [12].

First the code in the loop is pre-partitioned into a given number of clusters (two in this case) using a performance-centric method such as BUG. No restrictions on memory accessess are enforced. This results in the dataflow graph (DFG) with the partition shown by the dotted line in Figure 2(a). This initial partition places the accesses to the virtual memories containing arrays A, B, and C onto one cluster, while accesses to arrays D and E have been placed onto the other cluster.

Next, the memory synthesis phase considers combining the virtual memories into physical memories and assigning those memories to clusters. The best combination of virtual memories into physical memories is determined using the method detailed in Section 4. Combining two virtual memories into a physical memory on a particular cluster, may result in the insertion of one or more intercluster moves. For example, if arrays C and E were combined into a single physical memory on cluster 2, a move would be required to transfer the result of $LD(C[i])$ to the consumer as shown in Figure 2(a). The synthesis phase makes sure that the number of such extra intercluster moves do not exceed the global limit on intercluster move bandwidth.

After the memory synthesis phase, each local memory has been allocated to a physical memory on some cluster, as shown in Figure 2(b). In the figure, physical memory *C2_LM1* refers to the first local memory on the second cluster. Arrays C and E have been combined because both are of narrow bitwidth, and they have been allocated to the second cluster along with array D. The remaining arrays have been combined on the first cluster. Cluster assignment of the memories implies all load/stores to those arrays must be assigned to the home clusters. With this information, a final partitioning of the code is performed, taking into account the now-fixed load and store operations. This gives the partition shown in the DFG in Figure 2(c). Memory cost was minimized when combining virtual memories into physical memories, while performance was unaffected as the additional intercluster move was not on a critical path.

4 Local Memory Synthesis

As described in Section 3, physical memories are realized by combining one or more virtual memories together. Each virtual memory corresponds a single data structure in the application. In general, the data structures could be arrays, sub-arrays (an array partitioned into multiple virtual

memories), or linked data structures like hash tables, trees, etc. Although the method can be applied in the general scenario, the current implementation is limited to arrays. Specifically, we assume that one complete array comprises each virtual memory. The objective of the process is to find the set of combinations that yields the minimum cost subject to achieving the desired loop execution throughput specified by II.

Consider virtual memories $v_1, v_2, ...v_n$. We are concerned with combining some of these virtual memories into a single physical memory. Combining two virtual memories has both cost and performance implications. If v_i and v_j are combined into a single physical memory P, then the size of P will be the sum of the sizes of v_i and v_j, and the width of P will be the maximum of the bitwidths of v_i and v_j. Suppose II is the initiation interval required for the loop nest under consideration and a_i static loads in the loop body access the data structure corresponding to virtual memory v_i. When v_i and v_j are combined into P, there will be $a_i + a_j$ accesses to P per II cycles. Therefore P should have at least $\lceil \frac{a_i+a_j}{II} \rceil$ ports to sustain II.

Formally, we define the assignment of a set of virtual memories $\{v_{i_1}, v_{i_2}, ...v_{i_m}\}$ to a physical memory by S_i. Thus,

$$Size(S_i) = \sum_{k=1}^{m} Size(v_{i_k}), \quad Width(S_i) = \max_{1 \le k \le m} Bitwidth(v_{i_k}), \quad Ports(S_i) = \left\lceil \frac{\sum_{k=1}^{m} a_k}{II} \right\rceil$$

As described in Section 3, combining virtual memories also has an effect on the number of intercluster moves. Suppose virtual memories v_i and v_j are combined together into a single physical memory. The clustering algorithm is now forced to assign the static loads/stores corresponding to v_i and v_j to the same cluster. However, it could assign the uses of these static loads and the address computations feeding these loads/stores to a different cluster resulting in intercluster moves to transfer values. We denote the number of such intercluster moves required by $ICM_{(S,j)}$, when virtual memories $S = \{v_{i_1}, v_{i_2}, ...v_{i_m}\}$ are combined into a single physical memory, which resides in cluster j. Given II, combining virtual memories in S into a single physical memory on cluster j will cause $\lceil \frac{ICM_{(S,j)}}{II} \rceil$ intercluster moves in the steady state. Suppose MAX_{ICM} is the global limit on the number of intercluster moves allowed per cycle (referred to as move bandwidth). Then, we have the following constraint on combining virtual memories:

$$\max_j \left(\left\lceil \frac{ICM_{(S,j)}}{II} \right\rceil \right) \le MAX_{ICM}$$

Also, there is a constraint imposed by the added latency of the intercluster moves: the length of any recurrence cycle may not exceed the target II. This is enforced by first enumerating all the recurrence cycles in the loop. The lengths of the recurrence cycles for each physical memory realization are calculated taking into account the cluster assignments and path length increases due to required intercluster moves. Now, when some recurrence cycle length exceeds II, we forbid those combinations of virtual memories.

The aim now is to get an assignment of virtual memories to physical memories, so that the total cost is minimized while the intercluster bandwidth constraints are satisfied. Consider $2^S = \{S_1, S_2, ...S_{2^n-1}\}$, the set of all subsets of S (except the null set ϕ). Each subset S_i represents a virtual memory combination possibility, i.e., $S_i = \{v_{i_1}, v_{i_2}, ...v_{i_m}\}$ indicates that virtual memories $v_{i_1}, v_{i_2}, ...v_{i_m}$ are assigned to the same physical memory on a particular cluster. The problem now reduces to choosing a set of subsets from 2^S and a cluster assignment for each subset, such that each of $v_i \in S$ appears in exactly one of the chosen subsets, and the overall cost is minimized. This is the same as the well known set partitioning problem [1], and we solve it with an integer linear programming [22] formulation.

4.1 ILP Formulation

The model we assume for this formulation is that, each cluster has $P \times n$ local memories in it, where n is the number of virtual memories, and P is the maximum number of ports per memory that we want to consider. In each cluster, first n of the local memories have one port, next n

have 2 ports, next n have 3 ports, and so on, while the last n local memories have P ports. Now we consider assignment of the virtual memories $v_1, v_2, ...v_n$ to one of these local memories in some cluster. Therefore, the cost of a local memory can be calculated by knowing which virtual memories got assigned to it. If no virtual memories are assigned to a local memory, the local memory's cost automatically becomes zero.

We introduce a set of 0-1 variables $x_{i,j,k,l}$ for denoting the assignment of a virtual memory to some local memory. $x_{i,j,k,l} = 1$ implies that virtual memory v_i gets assigned to jth local memory with k ports, on cluster l, and $x_{i,j,k,l} = 0$ means, v_i was not assigned to that local memory. Every virtual memory should get assigned to exactly one local memory, which leads us to the following constraint.

$$\sum_{j=1}^{n}\sum_{k=1}^{P}\sum_{l=1}^{c} x_{i,j,k,l} = 1, \quad 1 \leq i \leq n$$

where c is the number of clusters in the machine. As described in Section 3.2, a performance centric clustering algorithm partitions the DFG corresponding to the innermost loop. This partition results in a certain number of intercluster moves for the entire loop, referred to as ICM_{total}. Now, assigning a virtual memory to a local memory on a particular cluster could cause a change in the total number of intercluster moves. This is because, the static loads and stores corresponding the virtual memory could have been initially assigned to different clusters. Assigning it to a local memory on a cluster now forces the static loads and stores to be assigned to that cluster, which may necessitate additional intercluster moves. The change in number of intercluster moves corresponding to assigning virtual memory v_i on cluster l is called ΔICM_l. The modulo scheduled loop can sustain only $\lceil \frac{MAX_{ICM}}{II} \rceil$ intercluster moves per cycle in the steady state. In other words, the total number of intercluster moves in the entire loop body should be less than $MAX_{ICM} \times II$. We model this constraint as below.

$$\left(ICM_{total} + \sum_{i=1}^{n}\sum_{l=1}^{c} \left[\Delta ICM_{i,l} \times \sum_{j=1}^{n}\sum_{k=1}^{P} x_{i,j,k,l} \right] \right) \leq MAX_{ICM} \times II$$

To calculate the cost of a local memory j with k ports, located on cluster l, we introduce variables to denote the size and bitwidth of those local memories. The size of a local memory j with k ports, located on cluster l is given by

$$s_{j,k,l} = \sum_{i=1}^{n} x_{i,j,k,l} \times Size(v_i), \quad 1 \leq j \leq n, 1 \leq k \leq P, 1 \leq l \leq c$$

Since the bitwidth of a local memory is the maximum bitwidth of all virtual memories assigned to it, we model the bitwidth using following constraints.

$$b_{j,k,l} >= x_{i,j,k,l} \times Bitwidth(v_i), \quad 1 \leq i, j \leq n, 1 \leq j \leq P, 1 \leq l \leq c$$

Suppose $Access(v_i)$ is the number of static accesses to virtual memory v_i in the loop body. The total number of accesses to a local memory with j ports should be less than j. This is modeled as

$$\sum_{i=1}^{n} x_{i,j,k,l} \times Access(v_i) \leq j \times II, \quad 1 \leq j \leq n, 1 \leq k \leq P, 1 \leq l \leq c$$

The cost of a local memory with k ports can be expressed as a linear function of its size and bitwidth as follows.

$$cost_{j,k,l} = X_k \times s_{j,k,l} + Y_k \times b_{j,k,l} + Z_k, \quad 1 \leq j \leq n, 1 \leq l \leq c$$

where X_k, Y_k, and Z_k are constants for all memories with k ports synthesized in a particular technology. These constants are obtained prior to solving the ILP by running the SRAM generator scripts from Artisan Inc. Thus, the integer linear program can be stated as, "Minimize the overall cost of local memories, given by

$$\sum_{j=1}^{n}\sum_{k=1}^{P}\sum_{l=1}^{c} cost_{j,k,l}$$

subject to constraints above".

Subset	Size	Wsize	# acc	Δicm	Ports	Cost
A	64	32	1	0	1	0.040
B	64	32	1	0	1	0.040
C	64	8	1	0	1	0.013
D	64	32	2	0	1	0.040
E	64	8	1	0	1	0.013
A,B	128	32	2	0	1	0.047
A,C	128	32	2	0	1	0.047
A,D	128	32	3	+2	2	0.070
A,E	128	32	2	+1	1	0.047
B,C	128	32	2	0	1	0.047
B,D	128	32	3	+2	2	0.070
B,E	128	32	2	+1	1	0.047
C,D	128	32	3	+2	2	0.070
C,E	128	8	2	+1	1	0.016
D,E	128	32	3	0	2	0.070

Subset	Size	Wsize	# acc	Δicm	Ports	Cost
A,B,C	192	32	3	0	2	0.070
A,B,D	192	32	4	+2	2	0.070
A,B,E	192	32	3	+1	2	0.070
A,C,D	192	32	4	+2	2	0.070
A,C,E	192	32	3	+1	2	0.070
A,D,E	192	32	4	+3	2	0.070
B,C,D	192	32	4	+2	2	0.070
B,C,E	192	32	3	+1	2	0.070
B,D,E	192	32	4	+3	2	0.070
C,D,E	192	32	4	+3	2	0.070
A,B,C,D	256	32	5	+2	3	∞
A,B,C,E	256	32	4	+1	2	1.104
A,B,D,E	256	32	5	+3	3	∞
A,C,D,E	256	32	5	+3	3	∞
B,C,D,E	256	32	5	+3	3	∞
A,B,C,D,E	320	32	6	+3	3	∞

Table 1. Cost estimates of possible groupings of arrays into physical memories for the example loop.

4.2 Example

The example in Figure 2 is examined in more detail to illustrate the memory synthesis process. Assume the II is two. The loop in the example has five arrays, therefore 31 groupings of the arrays into physical local memories are possible.

Table 1 shows the cost of each combination of the arrays into a physical memory. Note that array D is accessed twice in the loop and the rest of the arrays are accessed once. Combining two arrays into a single physical memory causes the total number of accesses to that memory to be the sum of accesses to the arrays assigned to it. For example, combining arrays A and B causes the number of accesses to be 2. The number of accesses to the physical memory determines its port requirements. As described before, the number of ports required on the physical memory is equal to $\lceil \frac{\#accesses}{II} \rceil$. Also, as shown in Figure 2, the number of intercluster moves incurred in the initial clustering phase is one, because the pre-partitioning (dotted line in Figure 2(a)) cuts one data flow edge. Combining two arrays together changes the number of intercluster moves, since the load/store operations corresponding to all arrays assigned to the same physical memory have to be assigned to the same cluster. Table 1 shows sizes, wordsizes, the number of ports required, the change in intercluster moves, and the actual cost of the physical memories corresponding to different combinations of arrays. Note that, when the port requirement is more than 2, we assume that the cost of the memory is ∞.

If each of the arrays were assigned to individual physical memories, then the overall area would be $0.040 + 0.040 + 0.013 + 0.040 + 0.013 = 0.146mm^2$. However, the ILP solver comes up with a better grouping of arrays. Arrays A and B are assigned to the same physical memory, which has an area of $0.047mm^2$, arrays C and E are assigned to the same physical memory which has an area of $0.016mm^2$, and array D is assigned to a single physical memory which has an area of $0.040mm^2$. Thus the overall area of the synthesized memory system is $0.047 + 0.016 + 0.040 = 0.103mm^2$.

5 Experimental Evaluation

Methodology. Our system was implemented using the Trimaran toolset [27], a retargetable compiler framework for VLIW/EPIC processors. We ran our experiments on a set of DSP kernels which are loop oriented and access many arrays in their inner loop. The inner loop bodies were pre-partitioned using the BUG algorithm [9] to compute the number of intercluster moves. Then, we determine the configuration of the local memories and assignment of arrays to them, by forming an integer linear program and solving it using **lp_solve** [3]. The area estimates for different memory configurations were obtained by running the SRAM generator scripts from ARTISAN Inc. We use the BUG algorithm to cluster the loop body again, and finally modulo schedule [18] the code. Note

Benchmarks	Achieved II							
	BW=2		BW=3		BW=4		BW=5	
	PRE	NONE	PRE	NONE	PRE	NONE	PRE	NONE
channel	14	20	10	13	7	10	6	8
huffman	20	28	14	19	10	14	8	12
LU	3	5	2	3	2	3	1	2
lyapunov	5	10	3	7	3	5	2	4
Avg reduction	37%		35%		33%		40%	

Table 2. Achieved II for several DSP kernels.

that the entire process takes the target II as input. We repeat the process for different IIs.

Results. Figure 3 shows the plots of area versus target IIs for four DSP kernels. There are two sets of plots for each benchmark, one where pre-partitioning was performed, and another where no pre-partitioning was performed. Each plot has four curves (black lines) corresponding to the synthesized memory configurations with intercluster move bandwidth limitations of 2, 3, 4, and 5 respectively. The corresponding max values (gray lines) show the combined areas where every array is in a single physical memory. Intelligent combining of memories reduces the overall area by up to 59% for LU. The intercluster move bandwidth does not affect the overall trend of the curve for most benchmarks. This is due to the fact that lower bandwidths do not constrain combination of arrays for these applications.

For all benchmarks, the cost savings achieved with intelligent combining over the naïve scheme is larger at higher IIs. This is because at lower IIs, the cost of additional memory ports constrains the number of combinations of virtual memories and thus the potential cost savings. Conversely, at higher IIs, many combinations are possible, which leads to large cost savings. Comparing the plots with and without pre-partitioning for each benchmark reveals little difference. This shows that the pre-partitioning did not impose significant restrictions on cost savings achieved through aggressive combining.

Even though the cost savings are similar, the pre-partitioning phase has a significant effect on the achieved II. Table 2 shows the minimum achieved II for each application at varying intercluster move bandwidths (BW). For each BW, the table shows the minimum achieved II in two cases: one where pre-partitioning was performed (PRE), and one where no pre-partitioning was performed (NONE). Clearly, the II in the PRE case is always better than the II in the NONE case. This is because without the pre-partitioning phase, the memory synthesis phase performs aggressive combining of arrays into memory. This causes the later partitioning phase to assign all the static loads and stores corresponding to those arrays to the same cluster. This, in turn, causes more intercluster moves to be inserted into the body of the loop, thereby increasing the II. The pre-partitioning phase improves the achieved II by 37%, 35%, 33%, and 40% for intercluster move bandwidths of 2, 3, 4, and 5, respectively.

6 Conclusion

In this paper, an approach for synthesizing custom local memory architectures for clustered loop accelerators is proposed. Given a target performance constraint and intercluster communication bandwidth, the objective is to create an organization that is minimal in cost. Each program data structure is initially placed in its own virtual local memory. Memory synthesis consists of an integer linear program solver that finds the best combination of virtual memories for each physical memory. At higher target IIs, the memory synthesis is able to achieve up to 59% cost savings by intelligently combining arrays with compatible cost and access characteristics into a single memory. An initial pre-partitioning phase ensures that memory synthesis is cognizant of the preferred assignment of operations to clusters. The best achievable II for a fixed intercluster move bandwidth is reduced between 23% and 57% when pre-partitioning is used. By carefully considering the intercluster communication effects, more intelligent combining decisions can be made with pre-partitioning.

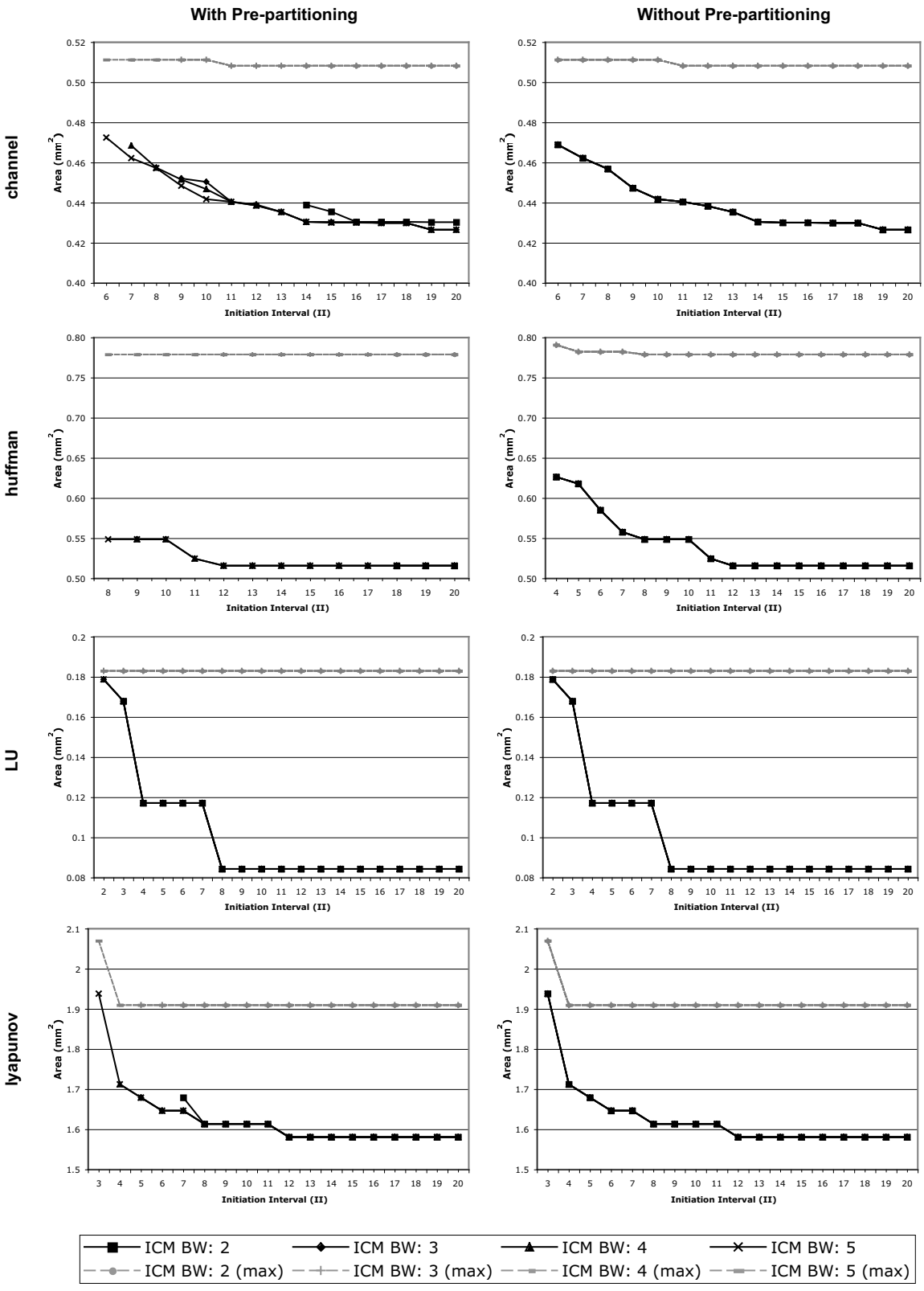

Figure 3. Local memory area vs. target II for several DSP kernels.

7 Acknowledgments

We thank the anonymous referees for their helpful comments and suggestions. We thank Alain Darte, Dushyant Sharma and Rob Schreiber for their useful suggestions to improve the ILP formulation. This research was supported in part by NSF Career Award grant CCF-0347411, NSF ITR grant CCR-0325898, ARM Limited, and equipment donated by Intel Corporation.

References

[1] E. Balas and M. Padberg. Set partitioning: A survey. *SIAM Review*, (18):710–760, 1976.

[2] L. Benini, L. Macchiarulo, A. Macii, E. Macii, and M. Poncino. From architecture to layout: Partitioned memory synthesis for embedded systems-on-chip. In *Proc. of DAC*, pages 784–789, 2001.

[3] M. Berkelaar. lp_solve: A Mixed Integer Linear Program Solver. Technical report, Eindhoven University of Technology.

[4] F. Catthoor et al. Global communication and memory optimizing transformations for low power signal processing systems. In *Proc. of Int. Workshop on Low Power Design*, pages 51–56, 1994.

[5] F. Catthoor et al. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[6] R. Cloutier and D. Thomas. The combination of scheduling, allocation and mapping in a single algorithm. In *Proc. of DAC*, pages 71–76, 1990.

[7] H. De Man et al. Architecture driven synthesis techniques for mapping digital signal processing structures into silicon. *Proc. of IEEE*, 78(2):319–335, Feb. 1990.

[8] N. Dutt, P. Panda, and A. Nicolau. *Memory Issues in Embedded System-on-chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1998.

[9] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[10] N. Holmes and D. Gajski. Architecture exploration for datapaths with memory hierarchy. In *Proc. of European Design and Test Conference*, pages 340–344, Mar. 1995.

[11] C. Huang, S. Ravi, A. Raghunathan, and N. Jha. High-level synthesis of distributed logic-memory architectures. In *Proc. of Int. Conf. on Computer-Aided Design*, 2002.

[12] IMEC. ATOMIUM Project. http://www.imec.be/atomium.

[13] M. Kandemir et al. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE TCAD*, 23(2):243–260, Feb. 2004.

[14] P. Marwedel. The MIMOLA system: Detailed description of the system software. In *Proc. of DAC*, pages 59–63, 1993.

[15] P. Panda and N. Dutt. Behavioral array mapping into multiport memories targetting low power. In *Proc. of Int. Conf. VLSI Design*, pages 268–272, Jan. 1997.

[16] P. Panda, N. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proc. of Int. Symposium on Systems Synthesis*, pages 90–97, 1997.

[17] L. Ramachandran, D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. In *Proc. of DAC*, pages 262–266, Mar. 1994.

[18] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc of Micro-27*, Dec. 1994.

[19] H. Schmit and D. Thomas. Synthesis of application-specific memory designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):101–111, Mar. 1998.

[20] R. Schreiber et al. High-level synthesis of nonprogrammable hardware accelerators. In *Proc. of the 2000 IEEE Intl. Conference on Application-Specific Systems, Architectures, and Processors*, Oct. 2000.

[21] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 31(2):127–142, Jun. 2002.

[22] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[23] L. Semeria, K. Sato, and G. De Micheli. Synthesis of hardware models in C with pointers and compelx data structures. *IEEE Transactions on VLSI Systems*, 9(6):743–756, Dec. 2001.

[24] J. Seo, T. Kim, and P. Panda. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *Proc. of 39th conference on Design Automation*, pages 609–611, 2002.

[25] J. Silva, F. Catthoor, D. Verkest, and H. De Man. Power-exploration through virtual memory management refinement. In *Proc of Int. Symp. on Low Power Electronics and Design*, Aug. 1998.

[26] D. Thomas. *Algorithmic and Register-transfer Level Synthesis*. Kluwer Academic Publishers, 1990.

[27] Trimaran. An Infrastructure for Research in ILP. http://www.trimaran.org.

[28] I. Verbauwhede, C. Scheers, and J. Rabaey. Memory estimation in high-level synthesis. In *Proc. of DAC*, pages 143–148, June 1994.

[29] Y. Zhao and S. Malik. Exact memory size estimation for array computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):517–521, Oct. 2000.