# High-Level Synthesis of Distributed Logic-Memory Architectures

Chao Huang[†], Srivaths Ravi[‡], Anand Raghunathan[‡], and Niraj K. Jha[†]
[†]**Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544**
[‡]**C&C Research Laboratories, NEC USA, Princeton, NJ 08540**
[†]**{chaoh, jha}@EE.Princeton.EDU** [‡]**{sravi, anand}@nec-lab.com**

*Abstract*— **With the increasing cost of global communication on-chip, high-performance designs for data-intensive applications require architectures that distribute hardware resources (computing logic, memories, interconnect, *etc.*) throughout a chip, while restricting computations and communications to geographic proximities. In this paper, we present a methodology for high-level synthesis (HLS) of distributed logic-memory architectures, *i.e.*, architectures that have logic and memory distributed across several partitions in a chip. Conventional HLS tools are capable of extracting parallelism from a behavior for architectures that assume a monolithic controller/datapath communicating with a memory or memory hierarchy. This work provides techniques to extend the synthesis frontier to more general architectures that can extract both coarse- and fine-grained parallelism from data accesses and computations in a synergistic manner. Our methodology selects many possible ways of organizing data and computations, carefully examines the trade-offs (*i.e.*, communication overheads, synchronization costs, area overheads) in choosing one solution over another, and utilizes conventional HLS techniques for intermediate steps.**

**We have evaluated the proposed framework on several benchmarks by generating register-transfer level (RTL) implementations using an existing commercial HLS tool with and without our enhancements, and by subjecting the resulting RTL circuits to logic synthesis and layout. The results show that circuits designed as distributed logic-memory architectures using our framework achieve significant (upto $5.31X$, average of $3.45X$) performance improvements over well-optimized conventional designs with small area overheads (upto $19.3\%$, $15.1\%$ on average).**

## I. INTRODUCTION

High-level synthesis (HLS) has been a topic of research for a long time, but has seen limited adoption in practice. We believe one of the key reasons has been that the quality of designs synthesized by HLS tools do not favorably compare against manual designs, given the wide range of advanced architectural tricks that experienced designers employ. While the basic concepts in HLS (*e.g.*, scheduling and resource sharing techniques) have been well established [1], [2], there is a need to extend the capabilities of HLS by importing various advanced techniques employed in the context of custom high-performance architectures. We explore one such technique in this work, namely the use of distributed logic-memory architectures.

Several important application domains, including digital signal, image and network processing, are characterized by large volumes of data access interleaved with computations. While several techniques have been developed to optimize memory access and the logic that implements the computations separately during HLS, significantly higher-quality designs can result if they are addressed in a synergistic manner, as shown in this paper.

### A. Related Work

Several techniques have been proposed to optimize memory architectures for high performance and low power. Since memory optimization has been explored in several different areas (*e.g.*, high-performance processors, embedded systems, distributed systems, parallel processing, *etc.*), which collectively represent a large body of research, a comprehensive review of related work is beyond the scope of this paper. We restrict our attention to memory optimization in the context of HLS, while acknowledging selected references from other fields that have motivated our work.

Reducing the memory size to exactly fit application requirements results in area, access time, and power benefits. Hence, techniques to estimate storage requirements from a behavior were proposed in [3], [4]. Behavioral transformation techniques to reduce memory requirements have been described in [5]. The problem of mapping (or binding) arrays in behavioral descriptions to one or more memories in RTL implementations has been addressed extensively in previous work. Some HLS systems map arrays in the behavioral description into a single monolithic memory [6], [7], [8], [9], while others take the opposite approach by mapping each array to a separate memory [10]. In general, neither extreme is the optimal solution, necessitating the use of a many-to-many mapping scheme. Techniques to reduce memory size while mapping multiple arrays to a single memory under performance constraints are presented in [11], [12]. Techniques for mapping multiple arrays to memories while reducing power consumption are described in [13]. The library mapping problem for memories, which consists of mapping generic memories to specific memory modules present in a library, is addressed in [14], [15]. Architectural exploration techniques for datapaths with hierarchical memory systems are presented in [16]. The work in [17] proposes a polyhedral analysis technique and an integer linear program (ILP) formulation to find the optimal memory architecture for a given performance constraint. Memory binding techniques for control-flow intensive behaviors are presented in [18]. All the works described above are restricted to static arrays in behavioral descriptions. Memory optimizations for more complex abstract data types, and dynamically allocated memory, are described in [19], [20]. Comprehensive data transfer and storage exploration methodologies that address most of the sub-problems described above, have been developed in the DTSE and ATOMIUM projects [21], [22]. Our work is complementary to most work on memory optimization during HLS, since it explores the partitioning of arrays (which are considered to be atomic in most of the above efforts) into smaller partitions, and does so in a manner that is synergistic with the partitioning of computations.

It bears mentioning that a significant body of research exists on functional partitioning of designs for HLS (see [23] for a good survey). These techniques typically focus on deriving irregular partitions. Our work focuses on regular partitions, by examining critical loops and deriving computation partitions through a tiling of the loop iteration space. Further, previous functional partitioning techniques perform computation partitioning without special consideration to memories, while the focus of our work is on joint computation and data partitioning to result in distributed logic-memory architectures.

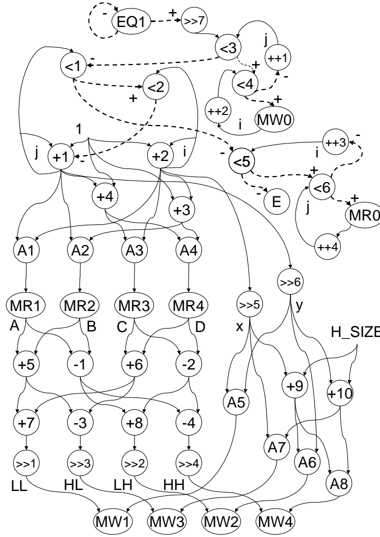
Our work draws upon techniques developed for joint partitioning of computations and data in domains such as high-performance microprocessors, parallel processing and distributed systems, and custom signal processing architectures, and attempts to adapt these techniques and integrate them into an HLS flow. The growing processor-memory gap and large disparity between on-chip and off-chip memory bandwidths prompted several prominent research efforts to integrate general-purpose processors and memory [24], [25], [26], [27]. In distributed systems, the high communication cost between processing elements has led to the development of techniques to partition computations and data to balance load and maximize performance [28], [29], [30], [31].

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-00-C-L516.

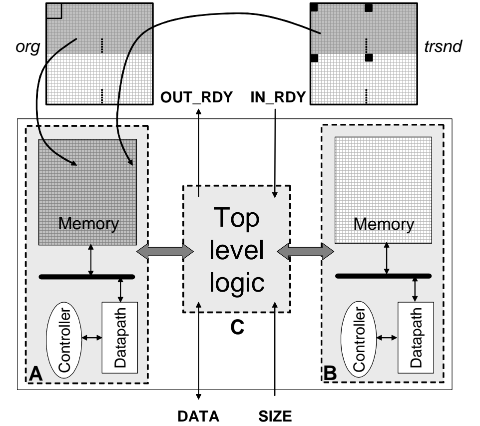
0-7803-7607-2/02/$17.00 ©2002 IEEE

```
#include <design_lib.h>
Wavelet(in IN_RDY,SIZE,inout DATA,out OUT_RDY)
{  if(IN_RDY==1){  //EQ1
     N=SIZE; H_SIZE=SIZE>>1; //>>7
   for(j=0, j<N, j++) //<4, ++2
     for(i=0, i<N, i++) //<3, ++1
        org[ADR(i,j)]=DATA; //MW0
   for(j=0; j<N; j+=2){    //<1, +1
     for(i=0; i<N; i+=2){//<2, +2
       A=org[ADR(i,j)];    //A1, MR1
       B=org[ADR(i+1,j)];  //+3, A2, MR2
       C=org[ADR(i,j+1)];  //+4, A3, MR3
       D=org[ADR(i+1,j+1)];//+3, +4, A4, MR4
      P=A+B+C+D;           //+5, +6, +7
      Q=A-B+C-D;           //-1, -2, +8
      R=A+B-C-D;           //+5, +6, -3
      S=A-B-C+D;           //-1, -2, -4
      x=i>>1; y=j>>1;      //>>5, >>6
      LL=P>>2; LH=Q>>2;    //>>1, >>2
      HL=R>>2; HH=S>>2;    //>>3, >>4
      trsnd[ADR(x,y)]=LL; //A5, MW1
      trsnd[ADR(x+H_SIZE,y)]=LH;//+9, A6, MW2
      trsnd[ADR(x,y+H_SIZE)]=HL;//+10, A7,MW3
      trsnd[ADR(x+H_SIZE,y+H_SIZE)];}}//+9,+10
                                      //A8,MW4
    OUT_RDY=1;
    for(i=0, i<N, i++)  //<5, ++3
      for(j=0, j<N, j++)  //<6, ++4
        DATA=trsnd[ADR(i, j)];} //MR0
```

(a)




(b)




(c)

Fig. 1. *Wavelet* design : (a) C Behavior, (b) CDFG representation, and (c) a two-way distributed logic-memory architecture

### B. Paper Overview and Contributions

In this work, we present techniques for HLS of distributed logic-memory architectures. The proposed methodology does not require any change to the core HLS algorithms. Hence, we believe that existing HLS flows can be easily adapted to take advantage of our techniques. Conventional HLS tools result in controller/datapath implementations that communicate with a monolithic memory, or at best a banked or hierarchical memory organization. We demonstrate that joint partitioning of the behavior's computations, and data that they operate on, results in architectures with superior performance characteristics. We also demonstrate that exploiting the performance improvement potential of distributed logic-memory architectures requires a systematic methodology to explore the complex tradeoffs involved. We propose an enhanced HLS flow that includes techniques to perform automatic data and computation partitioning, and discuss how these steps are interleaved with conventional HLS tasks such as scheduling and resource sharing. The enhanced HLS flow consists of the following steps: identification of critical code for partitioning (we focus on loops, since they account for the bulk of computations as well as data accesses in the application domains of interest), choice of the number of data-logic partitions, automatic techniques to identify partitions of the loop iteration space and the corresponding footprints in the data space, co-ordinated scheduling of the individual partitions to eliminate memory access conflicts introduced due to parallel accesses, and constrained resource sharing to result in distributed logic-memory implementations. We evaluate the proposed techniques using several example benchmarks in the context of a commercial design flow, and demonstrate that distributed logic-memory architectures can achieve significant performance improvements (upto $5.31X$, average of $3.45X$) over well-optimized designs generated by state-of-the-art HLS techniques.

## II. MOTIVATION: ILLUSTRATIONS AND ANALYSES

In this section, we illustrate the following issues using examples.

- Circuits designed using conventional HLS techniques are controller/datapath implementations that communicate with a single monolithic memory. For many applications, the use of banked memory or memory hierarchy eases data access bottlenecks to some extent. However, the performance gains are still inferior to what can be obtained when data and computations are partitioned in an integrated fashion (see Example 1).
- There are many possible ways of partitioning data and computations for a given circuit behavior. This means that the design space of possible partitioned implementations can be very large. Example 2 demonstrates that this design space necessitates many area and performance trade-offs mandating that (a) synthesis should work in conjunction with a design space exploration strategy, and (b) the overall flow must provide ways to model the "goodness" of a partition or a solution in terms of area, performance, communication overheads *etc.*
- Example 2 also point outs that scheduling for a distributed architecture requires existing HLS schedulers to be suitably enhanced to guarantee functional correctness of the overall system.

*Example 1:* Consider the behavior shown in Fig. 1(a) that describes a system implementing the discrete wavelet transform algorithm. The system takes as its inputs a flag IN_RDY that initiates data processing and an input/output channel DATA through which data of length SIZE are streamed. Input data are stored in an array $org$ on which the loop nest implementing the wavelet algorithm operates. The transformed image data are stored in the array $trsnd$, which is finally output through DATA with the output flag OUT_RDY set to 1. The control-data flow graph (CDFG) representation of this behavior is shown in Fig. 1(b). Each node in the CDFG corresponds to an operation in the behavior. For example, node $>> 5$ in the CDFG corresponds to the computation $x = i >> 1$ in the behavior. $MR0, \ldots, MR4$ and $MW0, \ldots, MW4$ represent memory read and write operations. Solid (dotted) edges in the CDFG represent data (control) dependencies between operations. A $+$ $(-)$ on a control edge denotes true (false).

Conventional HLS techniques, when applied to this CDFG, derive a controller/datapath RTL implementation of the circuit with the $org$ and $trsnd$ arrays stored in a monolithic memory block. An iteration of the schedule derived with a resource library consisting of two adders, two subtracters, three comparators, and two shifters is shown in Fig. 2(a). While addition and subtraction operations in this schedule take two cycles to complete, all other computations are single-cycle operations. Read (write) operations to the single-ported memory unit take an access time of four cycles. Fig. 2(a) shows a snapshot of simulating the resultant implementation using a $32 \times 32$ input image. The figure shows the schedule of computations in an iteration, wherein four pixels of data are processed in 38 cycles. Therefore, the steady-state performance of the circuit for applying the wavelet transform algorithm to a $32 \times 32$ image is 9,728 cycles.

The memory access times clearly contribute to a significant fraction of the overall execution times. Memory organization techniques such as memory banks are increasingly used in HLS to speed up data accesses to/from memory. Fig. 2(b) shows the schedule for a system with two memory banks such that odd and even columns of data are located in separate banks (distinguished by the shaded and non-shaded memory reads and writes in the figure). Reads or writes to the banked memory can begin two cycles after a previous memory operation. For a $32 \times 32$ input image, an iteration of wavelet now completes in 26 cycles, leading to a faster execution time of 6,656 cycles (a reduction of 32% compared to conventional HLS).

The performance of existing HLS solutions can be improved further if the knowledge of data accesses, computational patterns and the given resource constraints are used in an integrated manner during synthesis. Using the framework described in Section III, we can derive a high-performance architecture, as shown in Fig. 1(c). The architecture has three partitions (denoted A, B and C) with conventional HLS-style circuits (controller, datapath and memory are local to two partitions, while the third partition has a controller and datapath). Data in array $org$ are

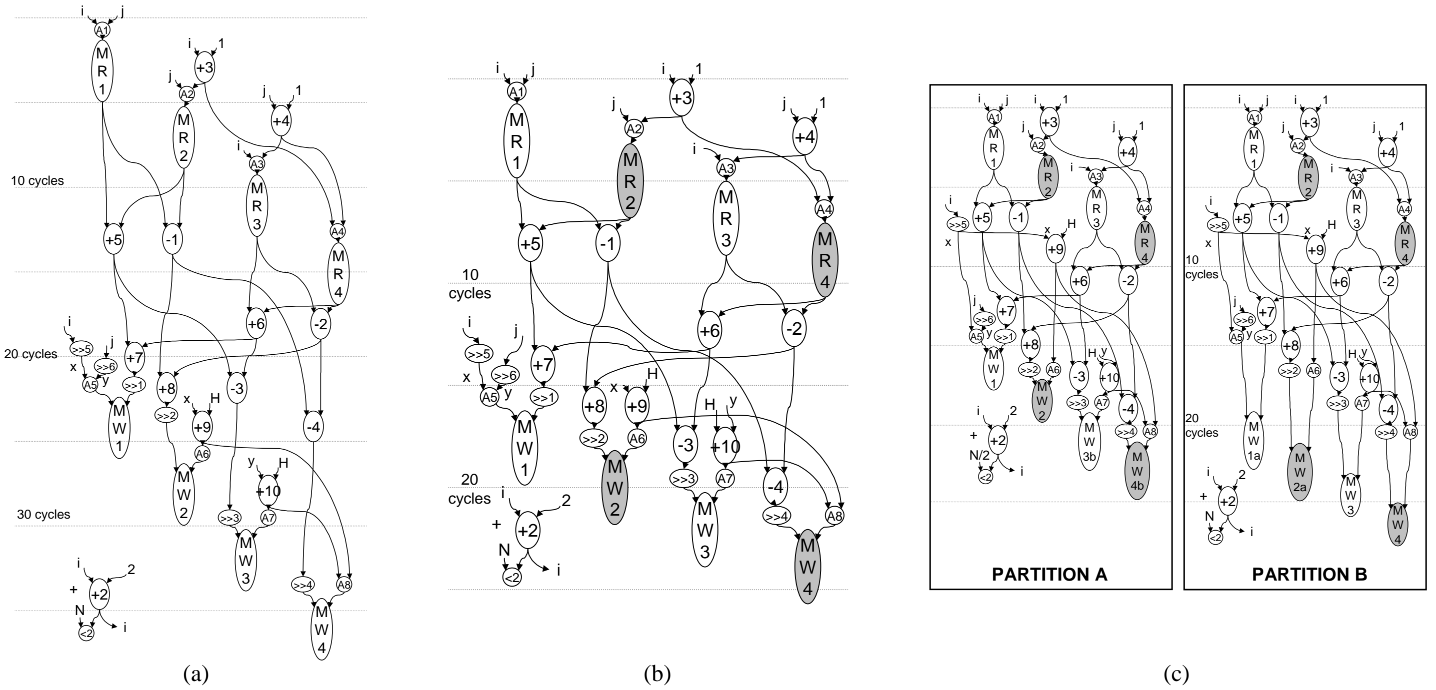


Fig. 2. Schedule snapshots for RTL implementations produced by (a) conventional HLS, (b) HLS enhanced for memory banks, and (c) our algorithm

first distributed among the two partitions, as shown in Fig. 1(c), by partition C. The controller in partition C regulates the functioning of the entire system and co-ordinates the individual schedules of the local controllers in each partition. Fig. 2(c) shows the snapshot of each partition's schedule derived in a co-ordinated manner using the scheduling formulation presented in Section III-B.3. The controller in each partition then executes its schedule on its local datapath, while reading/writing appropriately from its local memory or remote memory (memory local to another partition). For example, in the schedule for partition A, operations $MW1$ and $MW2$ are local memory accesses (which take three cycles), while $MW3b$ and $MW4b$ denote accesses to data stored in partition B (which take four cycles). Once the controller in each partition terminates its execution, the controller in partition C schedules the output of data from each local memory onto DATA.

The performance of the above system is better than both the conventional HLS and banked HLS cases. The execution time for the wavelet algorithm reduces to 3,584 cycles (2.71X and 1.86X speedups over the corresponding numbers in conventional and banked HLS solutions). In this way, the partitioned architecture of Fig. 1(c) distributes the available resources efficiently and extracts significant parallelism from the given behavior. ■

The next example details the issues and trade-offs involved in deriving a *good* partitioned implementation for a behavior.

*Example 2:* Figs. 3(a) and 3(b) show the main body of a behavior implementing the standard matrix multiplication algorithm and the corresponding CDFG representation, respectively. The behavior multiplies the elements of two matrices $P$ and $Q$, and stores the result in matrix $R$. HLS of the behavior in Fig. 3(a) results in a scheduling of computations as shown in Fig. 3(b). The schedule assumes a resource budget of (a) two Wallace-tree multipliers and adders with multiply (add) taking four (two) cycles, and (b) a single-ported memory with reads and writes taking four cycles. The snapshot shown is that of the schedule for the inner loop of the loop nest to multiply two $32 \times 32$ matrices. The schedule takes $267$ cycles to execute, leading to an overall execution time of 273,408 cycles for multiplying two $32 \times 32$ matrices. If synthesis must now determine a partitioned implementation that can better the above performance, it must consider several possible ways of partitioning data and computations, and evaluate and compare each candidate based on suitable cost metrics (area, performance, power, *etc.*). While handling such a vast design space efficiently is one problem, conventional HLS must be revamped to provide solutions that can address the following issues.

- Consider Fig. 4(a) which describes a possible choice of dividing matrices $P$, $Q$ and $R$ among two partitions A and B. $P1$ in the figure refers to data from the upper half of matrix $P$ ($P[1 \cdots N/2, 1 \cdots N]$), while $R1$ refers to data from the left quadrant of matrix $R$ ($R[1 \cdots N/2, 1 \cdots N/2]$) (assume $N$ is even), and so on. Data are organized among the two partitions so that partition A computes $R1$ and $R2$, while partition B computes $R3$ and $R4$. Fig. 4(c) shows the corresponding schedules for computations in the two partitions determined by conventional HLS schedulers. The schedules show that the computations of $R1$ and $R4$ happen concurrently in each partition, followed by those of $R2$ and $R3$. Observe that the memory reads and writes for computing $R1$ and $R4$ require accesses to data local to each partition ($P1$ and $Q1$ for $R1$, and, $P2$ and $Q2$ for $R4$). However, data for computing $R2$ or $R3$ require accesses from both partitions. Fig. 4(c) shows that the schedule for computing $R2$ requires a local access ($MR1$) for an element from $P1$ and a remote access ($MR2b$) for an element from $Q2$. Likewise, the schedule for computing $R3$ makes a local access ($MR1$) for an element from $P2$ and a remote access ($MR2a$) for

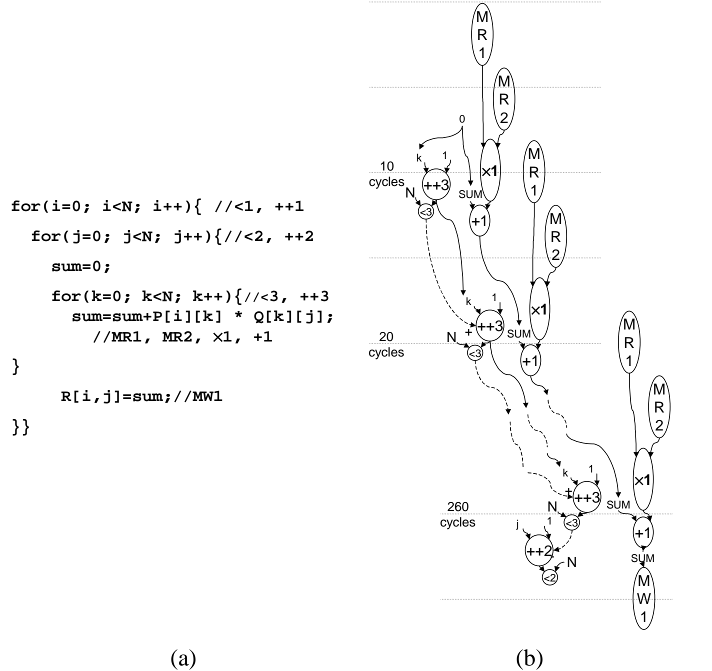

```
for(i=0; i<N; i++){ //<1, ++1
  for(j=0; j<N; j++){//<2, ++2
    sum=0;
    for(k=0; k<N; k++){//<3, ++3
      sum=sum+P[i][k] * Q[k][j];
        //MR1, MR2, ×1, +1
}
     R[i,j]=sum;//MW1
}}
```

Fig. 3. (a) C behavior for *Matrix*, and (b) schedule snapshot for conventional HLS

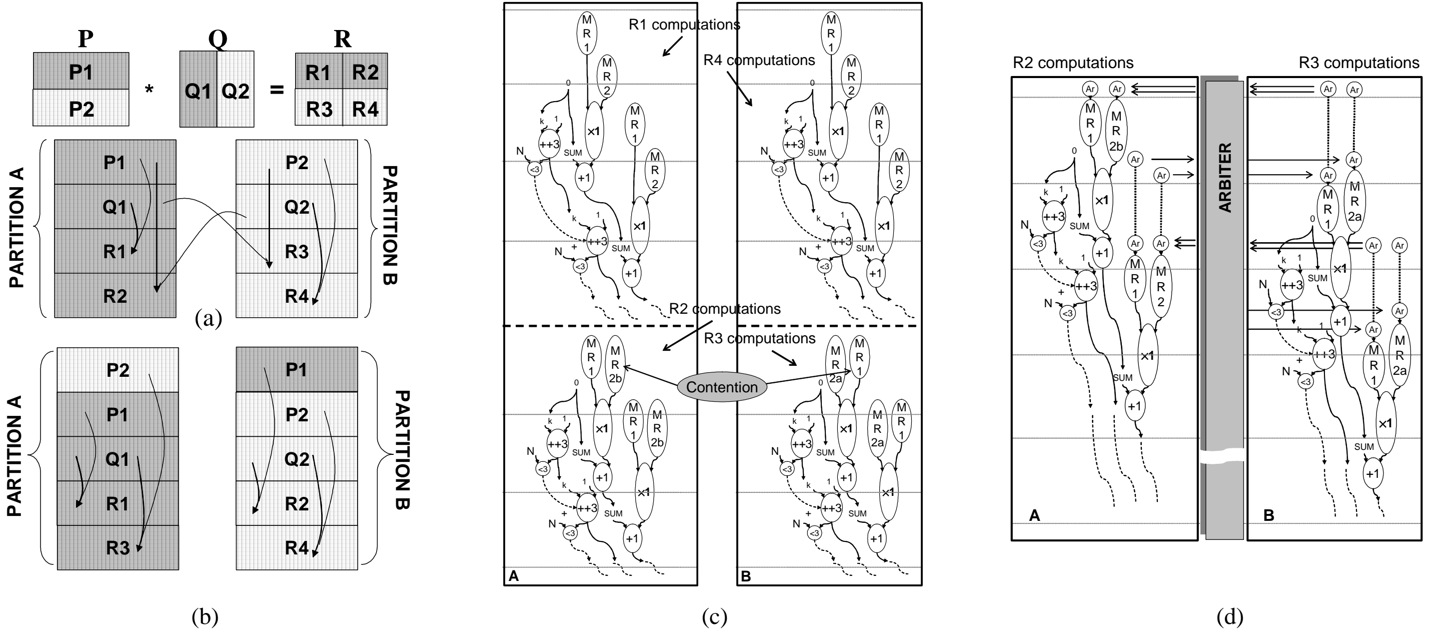



Fig. 4. (a) One possible data distribution for a two-way distributed logic-memory architecture, (b) another possible data distribution, (c) schedule with memory access bottlenecks, and (d) correct schedule with arbitration

an element from $Q1$. However, the schedule snapshots show that these accesses happen concurrently and cannot be satisfied by the single-ported memory in each partition, leading to a memory access bottleneck. Thus, schedules for each partition derived in isolation of one another cannot guarantee functional correctness of the overall system.

- A simple way to overcome the memory access bottleneck seen above is through the use of an arbiter to facilitate contention-free memory accesses. For example, access $MR2b$ in the schedule for computing $R2$ proceeds by first requesting the arbiter to check with the controller of partition B if such an access is feasible. If the arbiter finds the memory ready to service such a request, it allows partition A to access data from the local memory of partition B. Assuming a fair arbiter, the snapshots of memory accesses and computations for $R2$ and $R3$ are given in Fig. 4(d), with handshaking between the partitions as indicated. The schedules are functionally correct and the system execution time for computing $R2$ and $R3$ with arbitration is 75,624 cycles. Note that every memory access now requires an arbitration check and hence, the schedules of $R1$ and $R4$ also incorporate arbitration overheads and thus take 68,352 cycles. Therefore, the overall execution time of this configuration is 143,976 cycles.
- While the use of arbitration as above facilitates functionally correct schedules, it does not necessarily provide the best performing schedules. For example, Fig. 5 shows the schedules derived for each partition in a co-ordinated manner by our algorithm. No arbitration is required since remote memory accesses are made by a partition only when the other partition is free. Computing $R2$ and $R3$ now takes only 59,136 cycles. Schedules for $R1$ and $R4$ are also free of arbitration overheads and complete in just 51,968 cycles. Therefore, the overall execution time of this configuration is only 111,104 cycles. Note that unlike multi-processor systems which must be capable of executing many behaviors, ASIC designs require the system to be capable of executing a single behavior. Therefore, a good scheduling algorithm for distributed logic-memory architectures should not only provide functional correctness but also limit arbitration and communication overheads. Such a framework is provided in Section III.
- If data replication is possible, then partitioning must be able to factor in area costs. The performance and area of the RTL implementation change if we adopt the partitioning scheme presented in Fig. 4(b). Here, data from array $P$ are made available to both the partitions. Communication-free partitions are now feasible so that $R1$ and $R3$ are computed in partition A, and $R2$ and $R4$ are computed in partition B. The execution time of this configuration is 103,936 cycles. Note that, while this partitioning choice delivers enhanced performance and is communication-free due to data replication, it comes with area overheads (27.7% more area than the partitioning solution with communication).

■

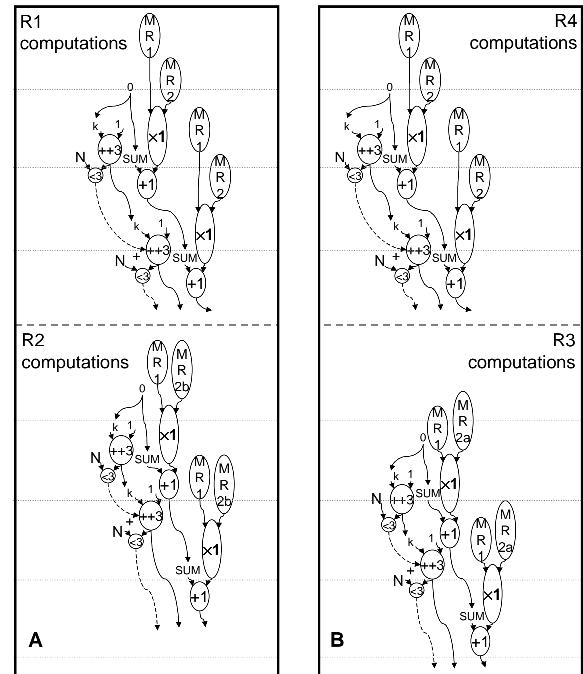



Fig. 5. Co-ordinated schedules for the two partitions

## III. METHODOLOGY AND ALGORITHMS

In this section, we describe our methodology for synthesizing multi-partitioned architectures. Section III-A presents an overview of the proposed framework, while Section III-B details the constituent steps.

### A. Overview

Fig. 6 outlines the flow for synthesizing circuits with multiple logic-memory partitions from a given circuit behavior. The inputs to the framework also include the design constraints (area and resource constraints) as well as the parameters to the different optimization steps. The shaded portions in the figure indicate steps where conventional HLS is applicable.

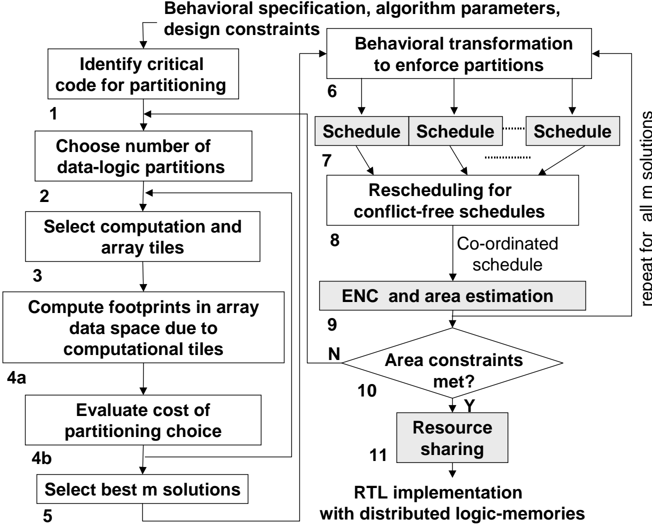



Fig. 6. HLS flow for obtaining RTL circuits with distributed logic-memory partitions

The algorithm performs a data dependence analysis of the behavior first, and demarcates critical portions of the code for further optimization (step **1**). This step typically identifies loop nests and regular portions of the code suitable for partitioning. Steps **2-4** then perform integrated computation and data partitioning for the targeted code. For a $K$-way partition (where $K$ can be specified), steps **3-4** consider candidate choices of partitioning data and computations $K$-way in order to find a good set of candidate partitions that can minimize the overall execution time as follows.

- A tiling of the iteration space (computations) requires some portions of the array data space to be read from and written to. Therefore, step **4a** first determines regions in different arrays (called *footprints*) referenced by each tile.
- For a given partitioning of the array data space, different portions of the footprint can physically belong to different partitions. Regions of a footprint present within a partition are local data accesses for the corresponding iteration space tile, while regions of a footprint outside the partition boundary are remote data accesses. The cost of a partitioning choice can now be formulated as a function of the local and remote accesses for each partition (step **4b**).

Partition space exploration in the above manner employs a fast but reasonably accurate estimation technique for computing the overall execution time. Based on the ranking of the partitioning candidates, the $m$ best solutions are selected (step **5**) for further analysis. Details of these steps are presented in Section III-B.1.

Steps **6-10** now perform a fine-grained evaluation of each solution. Step **6** first transforms the input behavior into communicating sub-behaviors to enforce the partitions on data and computations as determined by a solution (see Section III-B.2). Step **7** schedules the behavior identified for each partition using a conventional HLS scheduler. Since the schedules derived in this step for communicating partitions can contend for the same memory resource, step **8** performs an incremental rescheduling of each schedule to enforce conflict-free memory accesses (see Section III-B.3). The overall performance of the system can now be calculated based on the expected number of cycles (ENC) of the co-ordinated schedule (step **9**). High-level area estimates are also obtained at this stage. If area constraint is met, the framework exits successfully. Resource sharing can now be performed on the co-ordinated schedule with the restriction that no sharing is feasible between operations belonging to schedules in different partitions.

## B. Algorithms

In this subsection, we describe salient features of our algorithm. Section III-B.1 describes steps **4a-4b** of the algorithm, while Sections III-B.2 and III-B.3 detail steps **6** and **8**.

### *B.1 Footprints, partitions and their costs*

We first present the basic notation and terminology introduced in [28] to describe iteration space tiles, array references and basic footprints in a behavior with an example.

*Example 3:* Consider the two-level nested loop below, where, the array reference in the loop body is given by $A[\vec{g}(\vec{i})]$.

$$
\begin{aligned}
&for(i_1 = 0, i_1 < N, i_1++)\\
&\quad for(i_2 = 0, i_2 < N, i_2++)\\
&\qquad f(A[\vec{g}(\vec{i})]);
\end{aligned}
$$

Let the index function $\vec{g}(\vec{i})$ be affine with $\vec{i} = (i_1, i_2)$. In other words, let $\vec{g} = \vec{i}\mathbf{G} + \vec{a}$ with $\mathbf{G} = \begin{pmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{pmatrix}$, $\vec{a} = (a_1, a_2)$ where $\mathbf{G}$ is a $2 \times 2$ matrix with integer entries and $\vec{a}$ is the offset vector, an integer constant vector of length 2.

Consider a tiling of the iteration space such that the *basic tile* at the origin is a parallelogram given by matrix $\mathbf{L}^*$ below (any semi-open hyper-parallelopiped can be represented in this manner [28]). The row vectors of $\mathbf{L}^*$ are vertices of the basic tile.

$$\mathbf{L}^* = \begin{pmatrix} l^*_{11} & l^*_{12} \\ l^*_{21} & l^*_{22} \end{pmatrix}$$

Since footprints are regions of the array data space accessed by a tile, we can now determine the *basic footprint* matrix mapped by $\mathbf{L}^*$ for the reference $A[\vec{g}(\vec{i})]$ as follows.

$$
\begin{aligned}
\mathbf{D}^* = \quad \mathbf{L}^* \times \mathbf{G} &= \begin{pmatrix} l^*_{11}g_{11} + l^*_{12}g_{21} & l^*_{11}g_{12} + l^*_{12}g_{22} \\ l^*_{21}g_{11} + l^*_{22}g_{21} & l^*_{21}g_{12} + l^*_{22}g_{22} \end{pmatrix}\\
&= \begin{pmatrix} d^*_{11} & d^*_{12} \\ d^*_{21} & d^*_{22} \end{pmatrix}
\end{aligned}
$$

∎

The basic footprint determined in [28] gives only the shape and size of regions accessed in an array by a reference. However, the exact location of these regions needs to be determined to gauge how memory access patterns influence inter-partition communication. We illustrate our procedure for determining the desired regions using the following example.

*Example 4:* Matrix $\mathbf{D}^*$ in Example 3 defines the position and size of the basic footprint at the origin in the data array space shown in Fig. 7. Since the row vectors of $\mathbf{D}^*$ form a basis for the footprint, all the vertices of the footprint are simply linear combinations of the row vectors. In other words, they are given by set $FP^*$ below.

$$FP^* = \{(0,0), (d^*_{11}, d^*_{12}), (d^*_{21}, d^*_{22}), (d^*_{11} + d^*_{21}, d^*_{12} + d^*_{22})\}$$

In order to incorporate the effect of offset vector $\vec{a}$, consider Fig. 7. The actual footprint due to the array reference $A[\vec{g}(\vec{i})]$ (denoted $FP_1$) is a displacement of the basic footprint as shown. Then $FP_1$ is given by

$$
\begin{aligned}
FP_1 = \quad & \{\vec{v_{11}}, \vec{v_{12}}, \vec{v_{13}}, \vec{v_{14}}\}\\
= \quad & \{(a_1, a_2), (d^*_{11} + a_1, d^*_{12} + a_2), (d^*_{21} + a_1, d^*_{22} + a_2),\\
& (d^*_{11} + d^*_{21} + a_1, d^*_{12} + d^*_{22} + a_2)\}
\end{aligned}
$$

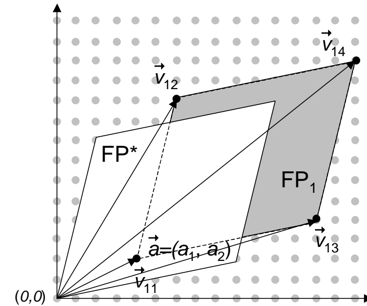



Fig. 7. Basic footprint $FP^*$ and footprint $FP_1$

For a given $K$-way tiling of the iteration space, all tiles are displacements of a basic tile through offset vectors $\{\vec{p_i}, i = 1, 2, \ldots, K\}$. In order to find the footprints associated with each tile, consider a tile $\mathbf{L}_i$ associated with an offset vector $\vec{p_i} = (p_{i1}, p_{i2})$ from the basic tile $\mathbf{L}^*$. Any point $\vec{i} = (i_1, i_2)$ in $\mathbf{L}^*$ accesses memory data at locations given by

$$\vec{i}\mathbf{G} + \vec{a}$$

The corresponding point $\vec{i} + \vec{p_i}$ in iteration tile $\mathbf{L_2}$ accesses the memory data at

$$(\vec{i} + \vec{p_i})\mathbf{G} + \vec{a} = (\vec{i}\mathbf{G} + \vec{a}) + \vec{p_i}\mathbf{G}$$

The above expression shows that the footprint of a tile $\mathbf{L}_i$ has (a) shape and size identical to the basic footprint, and (b) is associated with an offset vector $\overrightarrow{q_i} = \overrightarrow{p_i}\mathbf{G} = (q_{i1}, q_{i2})$ in the array data space. Thus, for any $FP_i$, the vertices of the footprint can be computed by adding the offset vector $\overrightarrow{q_i}$ to the vertices of the basic footprint $FP^*$.

$$\begin{aligned} FP_i = \; & \{\overrightarrow{v_{i1}}, \overrightarrow{v_{i2}}, \overrightarrow{v_{i3}}, \overrightarrow{v_{i4}}\} \\ = \; & \{\overrightarrow{v_{11}} + \overrightarrow{q_i}, \overrightarrow{v_{12}} + \overrightarrow{q_i}, \overrightarrow{v_{13}} + \overrightarrow{q_i}, \overrightarrow{v_{14}} + \overrightarrow{q_i}\} \\ = \; & \{(a_1 + q_{i1}, a_2 + q_{i2}), (d^*_{11} + a_1 + q_{i1}, d^*_{12} + a_2 + q_{i2}), \\ & (d^*_{21} + a_1 + q_{i1}, d^*_{22} + a_2 + q_{i2}), \\ & (d^*_{11} + d^*_{21} + a_1 + q_{i1}, d^*_{12} + d^*_{22} + a_2 + q_{i2})\} \end{aligned}$$

In this way, for a given $K$-way tiling of the iteration space, we can determine the regions $\{FP_i, i = 1, 2, \ldots, K\}$ corresponding to the footprints due to all the tiles. ∎

The footprints determined above demarcate the different regions of the array accessed by the behavior. In a partitioned architecture, each partition is associated with a fraction of each array and, hence, different fragments of a footprint. The best partitioning choice would, therefore, maximize the area of a footprint that lies within the local memory to minimize inter-partition communication. We can formulate a cost function to evaluate a partitioning choice using the following example scenarios (assume that a partition takes $\alpha$ cycles for a single read/write to its local memory and $\beta$ cycles for the corresponding operation to the remote memories).

- If footprint $FP_i$ (corresponding to loop tile $\mathbf{L}_i$) completely lies in data partition $\mathbf{P}_i$, all the data accesses for iteration space $\mathbf{L}_i$ can be satisfied by the local memory in *subsystem i*. Because the footprint size is approximately the number of memory accesses, we have the cost of memory access for loop tile $\mathbf{L}_i$ as the following.

$$cost_i = \alpha \times area(FP_i \cap \mathbf{P}_i) = \alpha \times area(FP_i)$$

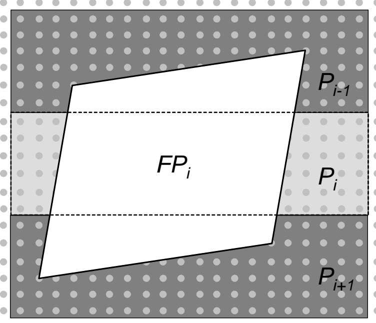



Fig. 8. Footprint $FP_i$ and the memory partitions

- If footprint $FP_i$ partially lies in data partition $\mathbf{P}_i$ as shown in Fig. 8, all the data accesses for iteration space $\mathbf{L}_i$ cannot be satisfied by the local memory in *subsystem i*. The overlapping area of footprint $FP_i$ and data partition $\mathbf{P}_i$, which approximately represents the local memory accesses, is given by

$$local_acc(FP_i) = area(FP_i \cap \mathbf{P}_i)$$

The remote accesses are given by the sum of the remote accesses made to partitions $\mathbf{P}_{i-1}$ and $\mathbf{P}_{i+1}$. In other words,

$$remote_acc(FP_i) = area(FP_i \cap \mathbf{P}_{i-1}) + area(FP_i \cap \mathbf{P}_{i+1})$$

Therefore, the estimated cost for the given choice of array partition and iteration tile is given by

$$\begin{aligned} cost_i = \; & \alpha \times area(FP_i \cap \mathbf{P}_i) \\ & + \beta \times (area(FP_i \cap \mathbf{P}_{i-1}) + area(FP_i \cap \mathbf{P}_{i+1})) \\ = \; & \alpha \times local_acc(FP_i) + \beta \times remote_acc(FP_i) \end{aligned}$$

In a $K$-way tiling, the total cost is therefore

$$Total\ Cost = \sum_{i=1}^{K} cost_i$$

Since the vertices of a footprint are known and the partition is given, the intersection region and its area can be computed from well-known techniques in computational geometry [32].

The cost function can now be formulated for the general case. Assume we have (a) a $K$-way partitioning of the system with the iteration space divided into $K$ tiles given by $\{L_i, i = 1, 2, \ldots, K\}$, (b) a set of data arrays $\{Array_v, v = 1, 2, \ldots, V\}$ in the original system, and (c) a set of data references $\{\overrightarrow{i}\mathbf{G}_{u_v} + \overrightarrow{a_{u_v}}, u_v = 1, 2, \ldots, U_v\}$ for each $Array_v$ in the nested loop. Let $FP_i^{(u_v)}$ (associated with loop tile $L_i$) represent the footprint of data reference $\overrightarrow{i}\mathbf{G}_{u_v} + \overrightarrow{a_{u_v}}$ to data array $Array_v$. Then the cost of accessing footprint $FP_i^{(u_v)}$ by $L_i$ is given by

$$cost_i^{u_v} = \alpha \times local_acc(FP_i^{(u_v)}) + \beta \times remote_acc(FP_i^{(u_v)})$$

Therefore, the total cost of this partitioned system is

$$Total\ Cost = \sum_{i=1}^{K} \sum_{v=1}^{V} \sum_{u_v=1}^{U_v} cost_i^{(u_v)}$$

In [28], cumulative footprints are introduced to aggregate footprints of a uniformly intersecting set of data references into a single bounding hyperparallelopiped. However, this aggregation leads to bad estimates of memory accesses whenever the offset vectors become large. Therefore, keeping the footprints disjoint as in our approach to compute local and remote accesses leads to more accurate cost estimates.

### *B.2 Behavioral Transformations for Computing Partitions' Schedules*

A candidate partition selected by step **5** can be further analyzed, once its effects are incorporated in the original behavior. The necessary transformations to prepare scheduler-ready behaviors corresponding to each partition are illustrated in Fig. 10 for *Wavelet*. Fig. 10(a) shows the modified behavior with sub-behaviors identified for two partitions, PARTITION_A and PARTITION_B, according to loop tiles given by regions $L_A = \{(0,0), (N/2-1, 0), (0, N-1), (N/2-1, N-1)\}$ and $L_B = \{(N/2, 0), (N-1, 0), (N/2, N-1), (N-1, N-1)\}$.

The array references in each sub-behavior must also be modified to indicate whether it is a local or remote access. Therefore, we transform every array reference as shown in Fig. 10(b) to first incorporate the effect of accesses to data located in the memories of different partitions. If $P1$ and $P2$ are the partitions on $trsnd$ and $FP1$ is the footprint corresponding to $L_A$ for reference $trsnd[ADR(x, y + H_SIZE)]$, then we include array index checks as shown. Now, the annotations can be correspondingly changed for HLS to operate correctly. Local access in the figure is denoted $MW3$ and remote access becomes $MW3b$. Note that in this case, $FP1 \cap P1$ is *null*. Hence, the corresponding statements can be eliminated. In this way, we can modify the different array references in each sub-behavior of *Wavelet* to obtain Fig. 10(c).

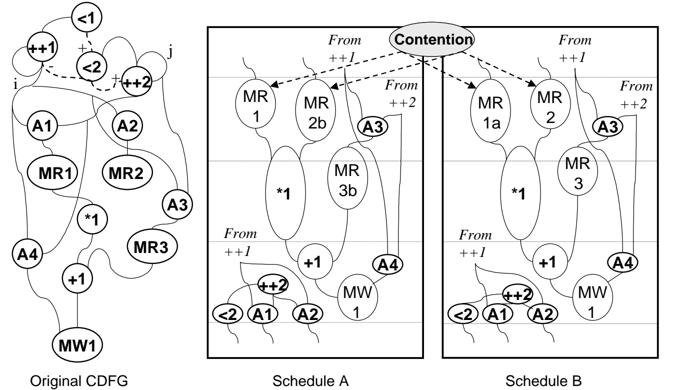



Fig. 9. Partitioned schedules and conflicts between memory accesses

### *B.3 Rescheduling for Conflict-free Accesses*

When schedules are derived by conventional HLS for the CDFGs corresponding to each sub-behavior (identified through the techniques in Sections III-B.1 and III-B.2), they can contend for access to the same memory resource in any state. Fig. 9 shows an example CDFG that has been bi-partitioned and the corresponding schedule snippets (schedules A and B) for executing an iteration of the partitioned behaviors. The schedules show that memory resource conflicts exist, for example, between a local access $MR1$ in schedule A and a remote access $MR1a$ in schedule B.

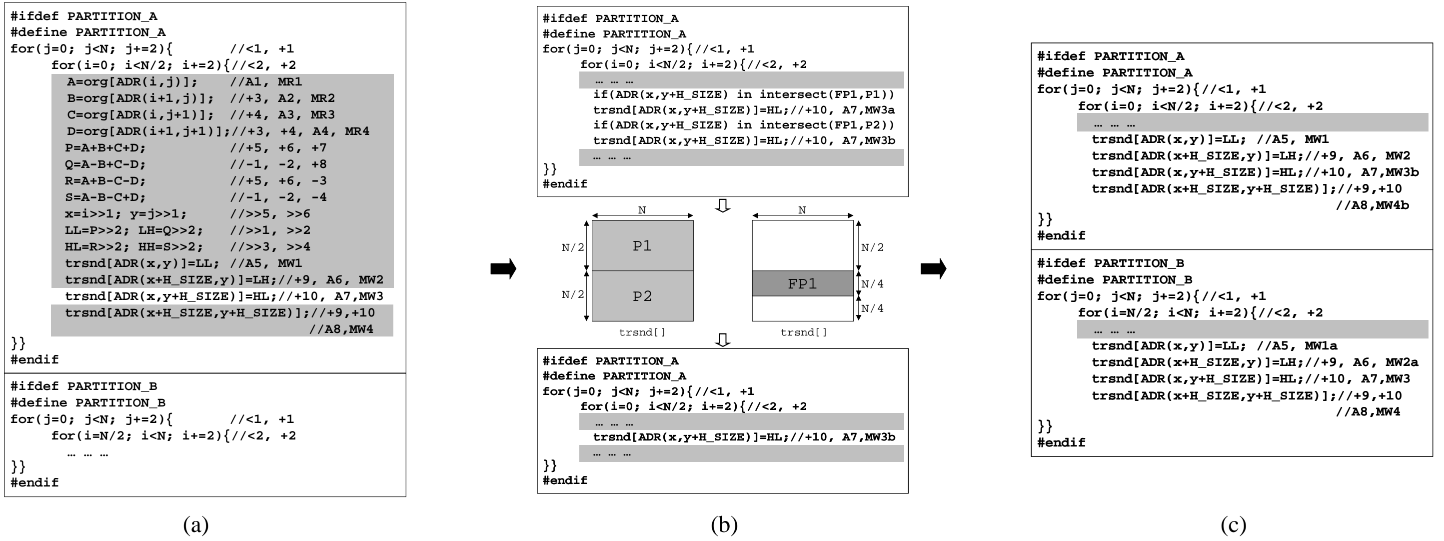



Fig. 10. *Wavelet* with (a) loops partitioned, (b) references modified and re-adjusted to introduce remote and local accesses, and (c) fully partitioned

While it is possible to compute the union of the two schedules and extract valid schedules for partitions A and B, such an approach is computationally inefficient. Therefore, we use a rescheduling strategy that incrementally modifies the two schedules to guarantee functional correctness without significantly losing the performance gains due to partitioning. We illustrate this strategy using the following example.

*Example 5:* The schedules in Fig. 9 are rescheduled upon detection of the indicated memory access conflicts as follows.

1. Each partition is given exclusive access to the contended memory resource. There are two scenarios possible based on the order in which this access is granted. For each scenario, the schedule of one partition remains unchanged, while the schedule of the other partition must be modified to reflect the delayed access. The delay is reflected in the schedule by introducing dummy primary inputs that arrive only when the conflicting memory access terminates in the other partition. Therefore, memory accesses and operations in their dependency fan-out are moved from their current states in the schedule to other states. This can possibly lead to resource conflicts in those states which lead to other operations moved to new states. In our example, Fig. 11(a) reflects this modification to the schedule of partition B through the addition of inputs $Ctrl1$ and $Ctrl2$ that arrive after the conflicting accesses complete in schedule A. Modified schedule B defers the scheduling of accesses $MR1a$ and $MR2$ and all operations dependent on them to later states as shown. Schedule A, however, remains unchanged.
2. The schedules derived in the above step can introduce new memory conflicts. In that case, we re-apply the above step to the modified schedules. For example, Fig. 11(a) shows conflicts between memory access $MR3b$ and $MR2$. Re-applying step 1 results in the modification in Fig. 11(b) which shows schedule A with a new control input $Ctrl3$ delaying $MR3b$ (schedule B remains unchanged now). Note that since the initial transformation can lead to a sub-optimal schedule ($+1$ can be scheduled with $A4$), we always perform incremental re-scheduling in conjunction with our transformation [33]. Incremental re-scheduling removes sub-optimalities by percolating operations wherever possible and, thus, compresses the schedule. This is shown in Fig. 11(c).

■

## IV. EXPERIMENTAL RESULTS

The techniques described in this paper were evaluated within the framework of an existing ASIC design flow. We applied our algorithm to several benchmark behaviors and were able to synthesize optimized circuits with multiple logic-memory partitions in each case. HLS of the input C behavior (with and without our techniques) was performed using a tool called *Cyber* [34]. The resultant RTL implementation was technology-mapped using *Design Compiler* [35] with the *TSMC* 0.25 micron standard cell-based library to obtain gate-level netlists. The *TSMC 0.25μm Process Single-Port SRAM Generator* [36] was used for all the memory blocks in the experiments. Gate-level functional simulation was performed using *SYNOPSYS VSS* and *Cyclone* [35] simulators. Finally, layouts of the original and optimized circuits were obtained using a suite of layout tools from Cadence (*CADENCE Silicon Ensemble*, *IC* and *LVD* [37]). The resulting gate-level circuits and layouts were compared with respect to the following metrics: *area* and *performance*. These metrics were extracted from the technology-mapped circuits and designer-provided testbenches. The results obtained are summarized in Fig. 12(a).

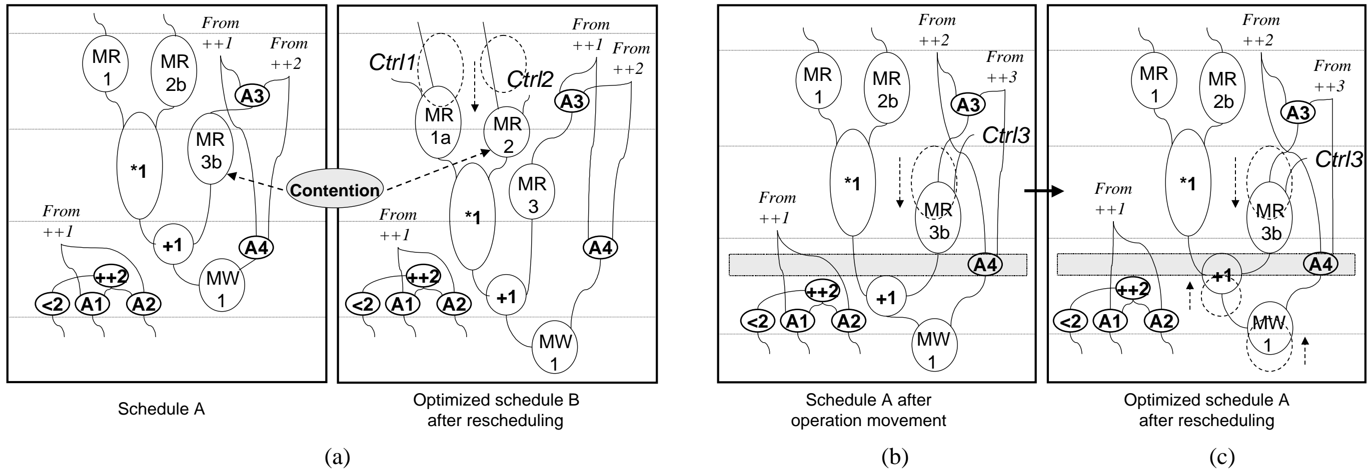
Of our benchmarks, *Wavelet* and *Matrix* were described in Section II. Finite impulse response (*FIR*) filter is a well-known signal processing benchmark. *Motion* is a behavior that performs video compression during video stream encoding. *YUV2RGB* implements image conversion from the YUV mode to RGB mode. *Edge* is a behavior performing edge detection in images.

In Fig. 12(a), major columns *Circuit, Area* and *Performance* represent the name of the behavior, area ($\times 10^6 DBU^2$) and performance expressed as the average execution time for processing $4K$ bytes of results data, respectively. Minor columns *Orig* and *Opt*, respectively, represent the original and the optimized partitioned systems. Columns A.O. and P.I. report the area overheads and performance improvements, respectively. Area overheads include the effect of replicating circuitry while partitioning physical memory, while performance improvements include the effects of rescheduling for conflict-free memory accesses across the partitions. A.O. is compared with the area constraint to determine the optimized partition.

The results presented in Fig. 12(a) describe the best performing partitioned architectures found by our algorithm for a target area constraint (20%) specified in each case. Two-way partitioned architectures were found for the first three examples, four-way for *Edge* and *YUV2RGB*, and eight-way for *Motion*. The latter applications have a high memory access to computation ratio and benefit from an increase in the number of partitions. However, this trend is not reflected in all the cases. For example, in $FIR$, the performance actually becomes worse as the number of partitions is increased from 8 to 16. Fig. 12(b) shows how the performance and area vary as the number of partitions is increased. The performance improvements begin to diminish as the remote accesses start dominating. Eventually, the benefits of partitioning get outweighed by the communication overheads. The area overhead varied from 10.9% to 19.3% for the examples. It includes the effect of laying out the complete system with memory blocks and routing effects.

## V. CONCLUSIONS

In this paper, we proposed a methodology for HLS of distributed logic-memory architectures. Our methodology includes techniques for synergistic partitioning of the computations and data associated with critical loops in the behavioral description, followed by constrained scheduling and resource sharing to result in the desired target architectures. Experiments with several benchmarks in the context of a state-of-the-art design flow demonstrated that the proposed techniques achieve significant improvements in performance under certain constraints. We believe that advanced HLS techniques, such as the ones proposed in this work, can bridge the performance gap between RTL circuits generated by HLS and

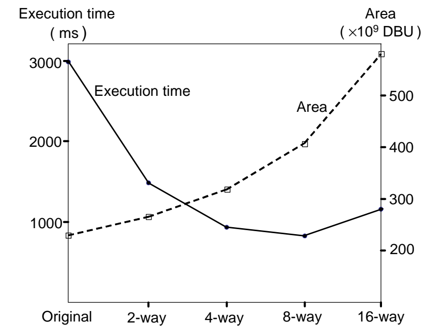



(a) (b) (c)

Fig. 11. Steps involved in rescheduling for conflict-free memory accesses

| Circuit | Area ($\times 10^6 DBU^2$) | | | Execution time ($ms$) | | |
|---|---|---|---|---|---|---|
| | Orig. | Opt. | A.O. (%) | Orig. | Opt. | P.I. |
| *Wavelet* | 353,499 | 399,807 | 13.1 | 233.47 | 86.01 | 2.71X |
| *Matrix* | 527,204 | 584,669 | 10.9 | 13,123.58 | 5,332.99 | 2.46X |
| *FIR* | 234,316 | 269,698 | 15.1 | 3,002.09 | 1,471.61 | 2.04X |
| *Edge* | 209,338 | 246,391 | 17.7 | 1,376.39 | 305.18 | 4.51X |
| *YUV2RGB* | 329,697 | 377,833 | 14.6 | 1,474.75 | 405.16 | 3.64X |
| *Motion* | 240,209 | 286,569 | 19.3 | 661.12 | 124.50 | 5.31X |

(a)




(b)

Fig. 12. (a) Area and performance comparisons for the original and optimized circuits, and (b) performance and area variations with different partitions for *FIR*

expert manual designs, and will promote the adoption of HLS tools in ASIC design flows for promising application domains such as multimedia processing and network processing.

## References


[1] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin, *High-level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Norwell, MA, 1992.
[2] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994.
[3] I. M. Verbhauwehede, C. J. Scheers, and J. Rabaey, “Memory estimation in high-level synthesis,” in *Proc. Design Automation Conf.*, June 1994, pp. 143–148.
[4] Y. Zhao and S. Malik, “Exact memory size estimation for array computations,” *IEEE Trans. VLSI Systems*, vol. 8, no. 5, pp. 517–521, Oct. 2000.
[5] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, “Global communication and memory optimizing transformations for low power signal processing systems,” in *Proc. Int. Wkshp. Low Power Design*, 1994, pp. 51–56.
[6] P. Marwedel, “The MIMOLA system: Detailed description of the system software,” in *Proc. Design Automation Conf.*, June 1993, pp. 59–63.
[7] H. De Man, F. Catthoor, G. Goossens, J. V. Meerbergen, J. Rabaey, and J. Huisken, “Architecture driven synthesis techniques for mapping digital signal processing structures into silicon,” *Proc. IEEE*, vol. 78, no. 2, pp. 319–335, Feb. 1990.
[8] R. Cloutier and D. Thomas, “The combination of scheduling, allocation, and mapping in a single algorithm,” in *Proc. Int. Symp. Microarchitecture*, Dec. 1996, pp. 126–137.
[9] N. Dutt, P. R. Panda, and A. Nicolau, *Memory Issues in Embedded Systems-on-chip: Optimizations and Exploration*, Kluwer Academic Publishers, Norwell, MA, 1998.
[10] D. Thomas, *Algorithmic and Register-transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publishers, Norwell MA, 1990.
[11] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, “An algorithm for array variable clustering,” in *Proc. European Design Automation Conf.*, Mar. 1994, pp. 262–266.
[12] H. Schmidt, “Synthesis of application-specific memory designs,” *IEEE Trans. VLSI Systems*, vol. 5, no. 1, pp. 101–111, Mar. 1997.
[13] P. R. Panda and N. D. Dutt, “Behavioral array mapping into multiport memories targeting low power,” in *Proc. Int. Conf. VLSI Design*, Jan. 1997, pp. 268–272.
[14] P. K. Jha and N. D. Dutt, “High-level library mapping for memories,” *ACM Trans. Design Automation of Electronic Systems*, no. 3, pp. 566–603, July 2000.
[15] O. Sentieys, D. Chillet, J. P. Diguet, and J. L. Phillipe, “Memory module selection for high-level synthesis,” in *Proc. VLSI Signal Processing IX*, Oct. 1996, pp. 273–282.
[16] N. D. Holmes and D. D. Gajski, “Architecture exploration for datapaths with memory hierarchy,” in *Proc. European Design & Test Conf.*, Mar. 1995, pp. 340–344.
[17] F. Balasa, *Background Memory Allocation for Multi-dimensional Signal Processing*, Ph.D. thesis, ESAT/EE Dept., K.U.Leuven, Belgium, 1995.
[18] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, “Memory binding for performance optimization of control-flow intensive behaviors,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1999, pp. 482–488.
[19] J. L. da Silva, F. Catthoor, D. Verkest, and H. De Man, “Power exploration for dynamic data types through virtual memory management refinement,” in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 1998.
[20] L. Semeria, K. Sato, and G. De Micheli, “Synthesis of hardware models in C with pointers and complex data structures,” *IEEE Trans. VLSI Systems*, vol. 9, no. 6, pp. 743–756, Dec. 2001.
[21] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*, Kluwer Academic Publishers, Norwell, MA, 1998.
[22] *ATOMIUM Project, IMEC*, http://www.imec.be/atomium.
[23] F. Vahid, “Techniques for minimizing and balancing I/O during functional partitioning,” *IEEE Trans. Computer-Aided Design*, vol. 18, no. 1, pp. 69–75, Jan. 1999.
[24] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, “Scalable processors in the billion-transistor era: IRAM,” *Computer*, vol. 30, no. 9, pp. 75–78, Sept. 1997.
[25] M. Oskin, F. T. Chong, and T. Sherwood, “Active pages: A computation model for intelligent memory,” in *Proc. Int. Symp. Computer Architecture*, June 1998, pp. 192–203.
[26] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, “Flexram: Toward an advanced intelligent memory system,” Oct. 1999.
[27] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart memories: A modular reconfigurable architecture,” in *Proc. Int. Symp. Computer Architecture*, June 2000, pp. 161–171.
[28] A. Agarwal, D. A. Kranz, and V. Natarajan, “Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors,” *IEEE Trans. Parallel & Distributed Systems*, vol. 6, no. 9, pp. 943–962, Sept. 1995.
[29] J.-A. M. Anderson, “Automatic computation and data decomposition for multiprocessors,” Tech. Rep. CSL-TR-97-719, Computer Systems Lab, EECS, Stanford Univ., Stanford, CA, Mar. 1997.
[30] A. W. Lim, *Improving Parallelism and Data Locality with Affine Partitioning*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, Aug. 2001.
[31] J. Ramanujam and P. Sadayappan, “Tiling multidimensional iteration spaces for multicomputers,” *J. Parallel & Distributed Computing*, vol. 16, no. 2, pp. 108–230, 1992.
[32] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry*, Springer-Verlag, 1997.
[33] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, “Percolation based synthesis,” in *Proc. Design Automation Conf.*, June 1990, pp. 444–449.
[34] K. Wakabayashi, *C-Based High-Level Synthesis System, “CYBER”-Design Experience-*, vol. 41, pp. 264–268, July 2000.
[35] *SYNOPSYS Design Compiler, VSS and Cyclone User Manual*, http://www.synopsys.com.
[36] *TSMC 0.25mm Process High-Density Single-Port SRAM (HD-SRAM-SP) Generator User Manual*, http://www.artisan.com.
[37] *Cadence Openbook SE 5.3, IC 4.4.5 and LVD 3.0*, http://www.cadence.com.