

# Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization

JASON CONG, WEI JIANG, BIN LIU, and YI ZOU, University of California, Los Angeles

Memory bottleneck has become a limiting factor in satisfying the explosive demands on performance and cost in modern embedded system design. Selected computation kernels for acceleration are usually captured by nest loops, which are optimized by state-of-the-art techniques like loop tiling and loop pipelining. However, memory bandwidth bottlenecks prevent designs from reaching optimal throughput with respect to available parallelism. In this paper we present an automatic memory partitioning technique which can efficiently improve throughput and reduce energy consumption of pipelined loop kernels for given throughput constraints and platform requirements. Also, our proposed algorithm can handle general array access beyond affine array references.

Our partition scheme consists of two steps. The first step considers cycle accurate scheduling information to meet the hard constraints on memory bandwidth requirements specifically for synchronized hardware designs. An ILP formulation is proposed to solve the memory partitioning and scheduling problem optimally for small designs, followed by a heuristic algorithm which is more scalable and equally effective for solving large scale problems. Experimental results show an average 6× throughput improvement on a set of real-world designs with moderate area increase (about 45% on average), given that less resource sharing opportunities exist with higher throughput in optimized designs. The second step further partitions the memory banks for reducing the dynamic power consumption of the final design. In contrast to previous approaches, our technique can statically compute memory access frequencies in polynomial time with little or no profiling. Experimental results show about 30% power reduction on the same set of benchmarks.

Categories and Subject Descriptors: B.5.2 [Register-Transfer-Level Implementation]: Design Aids—*Automatic synthesis*

General Terms: Algorithm, Design, Experimentation

Additional Key Words and Phrases: Behavioral synthesis, memory partition

## ACM Reference Format:

Cong, J., Jiang, W., Liu, B., and Zou, Y. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.* 16, 2, Article 15 (March 2011), 25 pages. DOI = 10.1145/1929943.1929947 <http://doi.acm.org/10.1145/1929943.1929947>

## 1. INTRODUCTION

Behavioral synthesis has emerged as a promising direction in EDA because of the exponentially increasing complexity in modern SoC designs. Behavioral synthesis transforms algorithmic description of behavior into hardware implementation. Typical applications for behavioral synthesis are data-intensive or computation-intensive kernels in multimedia processing, which require high throughput. Usually computation

---

A preliminary version of this article appeared as Cong et al. [2009].

This work is partly supported by GSRC, SRC Contract 2009-TJ-1879, and NSF grant CNS-0725354. Support from Magma and AutoESL Inc. is also gratefully acknowledged.

Authors' address: J. Cong, W. Jiang, B. Liu, and Y. Zou, Computer Science Department, University of California, Los Angeles, CA 90095; email: {cong,wjiang,bliu,zouyi}@cs.ucla.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2011 ACM 1084-4309/2011/03-ART15 \$10.00

DOI 10.1145/1929943.1929947 <http://doi.acm.org/10.1145/1929943.1929947>

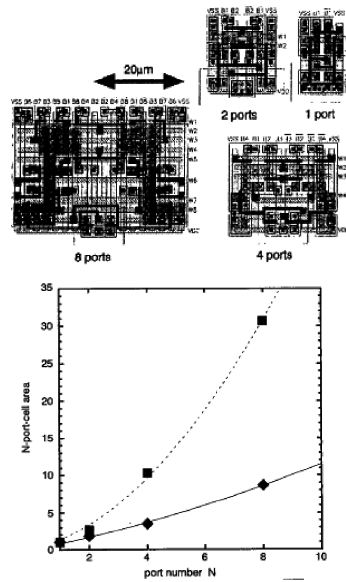


Fig. 1. Area of SRAM cells (taken from Tatsumi and Mattausch [1999]).

kernels are captured by perfect loop nests. Loop tiling is a common optimization technique that partitions a loop's iteration space into smaller chunks or blocks, so as to increase the parallelism inside one loop iteration, and improve data locality. However, memory bandwidth limits potential performance gain even with large parallelism available. For instance, in a pipelined design, the final throughput is limited by number of memory accesses divided by memory ports.

Memory bottleneck problems have drawn intensive interest, in both software and hardware designs. A natural way to tackle this problem is to increase memory ports in the final memory implementations. However, research [Tatsumi and Mattausch 1999] shows that the size of memory cells increases almost quadratically with port number  $N$  (shown in Figure 1), which limits the maximal memory port number in practical designs. Moreover, increasing memory ports is not feasible on FPGA platforms; for example, block RAMs on Xilinx Virtex systems<sup>1</sup> are restricted to two read-write ports. By duplicating the original memory  $N$  times, memory read bandwidth can be improved by  $N$  times. But, despite the large area overhead, the number of simultaneous memory writes cannot be increased because it is difficult to guarantee that all memories have synchronized data. A better approach is memory partitioning, which divides original memories into several banks. To take advantages of memory partitioning, compilers must analyze memory access patterns in original programs carefully, and balance them among different banks.

Data partitioning and loop partitioning have been studied intensively in the compiler domain for parallel computing on distributed systems. Ramanujam and Sadayappan [1991] and Anderson and Lam [1993] developed algorithms to generate communication-free data partitions for multi-computers with local memory. The approach in Agarwal et al. [1995] presented an optimal tiling and iteration/data space mapping algorithm even if communication-free partitioning solutions do not exist. In general, these algorithms attempt to exploit coarse-level parallelism to generate good data partitions that

<sup>1</sup><http://www.xilinx.com>

minimize remote data accesses. However, in hardware design, data accesses must be considered at cycle-accurate level to avoid excessive simultaneous accesses on the same memory. In behavioral synthesis, Gong et al. [2005] studied the memory partitioning problem based on the footprint of data access, where only either row-based partitioning or column-based partitioning can be supported. Another approach [Baradaran and Diniz 2008] studied the impact of data distribution, data duplication and scalar replacement for behavioral synthesis to reduce latency of loop bodies. Their work did not consider scheduling information in the data partitioning formulation. Our approach considers memory partitioning at the cycle-accurate level to meet throughput requirements on pipelined perfect loop nests.

Memory partitioning can also be applied for power optimization. As IC technology scales into the sub-100nm regime, power optimization has emerged as one of the utmost important tasks in SoC designs. Power consumption usually goes up with performance improvement; therefore, power optimization is considered in this paper as a secondary objective to compensate for the overhead brought by performance optimization. Memory partitioning techniques for power reduction have been studied [Luh and Kim 2004; Wang and Hu 2004; Poncino et al. 2000]. Luh and Kim [2004] and Wang and Hu [2004] assign discrete memory read/write operations to different memory banks for performance and power optimizations. Their formulations mainly work for data flow graphs where one memory access can be bound to only one memory bank statically. The approach in Poncino et al. [2000] generates an optimal memory partitioning scheme based on platform information and memory access frequencies. The access frequencies are assumed to be given in their formulation which is usually done by profiling. In our approach, each array access is modeled as a parameterized polytope, and array access profiles can be calculated by a polyhedron library to get an analytical solution with respect to loop bounds and other parameters. For multimedia programs, our approach usually requires no profiling (for affine array accesses) or lightweight profiling by using a parameterized polytope model. In addition, our approach can support more partitioning schemes.

In this article, a synthesis flow is proposed to solve the memory partitioning problem for throughput and power optimization. The goal of throughput optimization is to maximize concurrent memory accesses by partitioning; however, power optimization tries to shut down unused memory banks to save dynamic power consumption. These distinct objectives make it extremely difficult to tackle throughput and power optimization in one step. In our approach, the throughput optimization problem is solved at the first step, then results are passed to the power optimization step to further reduce the total energy consumption.

Our contributions include the following: (i) a behavioral synthesis flow which generates optimal memory partitioning solutions automatically for fast design space exploration. (ii) Unlike most previous work, our approach can handle array references beyond the affine form, such as indirect array accesses and nonaffine array subscriptions. (iii) Moreover, our partition scheme considers cycle-accurate scheduling information to meet the hard constraints on memory bandwidth requirements, specifically for synchronous hardware designs. (iv) An effective power optimization algorithm is proposed that considers more general partition schemes and requires less profiling effort. Compared to previous works, our approach requires only lightweight profiling, and can support more partitioning schemes for better results. These contributes are validated by our experimental results; more than  $15\times$  performance improvement and 60% power reduction can be observed on certain designs.

A preliminary version of this work was reported in Cong et al. [2009]. In this article, we proposed an integer-linear-programming (*ILP*) based algorithm to optimally solve the memory partitioning and scheduling together for small examples. And our

algorithm is extended to consider loop dependences as well for throughput optimization. Furthermore, an interactive modulo scheduling algorithm is adapted for the final loop pipelining following our memory partitioning step.

The remainder of the article is organized as follows. Section 2 describes the preliminaries, such as polytope model and loop pipelining. Section 3 gives a motivation example for our memory partitioning problem. Section 4 gives problem formulation of our memory partitioning problem, and Sections 5–7 present our proposed algorithms for solving this problem. Section 8 reports experimental results and is followed by conclusions in Section 9.

## 2. PRELIMINARIES

### 2.1. Polytope Model

Computation-intensive parts of applications are commonly captured by perfect loop nests. Without loss of generality, all loops are normalized to have unit step size. Loops and memory accesses can be modeled as polytopes on rational spaces.

*Definition 1.* The iteration space  $\mathbb{I}$  of a depth- $l$  loop nest is defined in the  $l$ -dimensional space, where each iteration corresponds to an integer vector identified by its index values  $I = (i_1, i_2, \dots, i_l)$ .

*Definition 2.* Similarly, an  $n$ -dimensional array is defined by an array space  $\mathbb{A}$ , and each array access at a certain iteration is determined by an integer tuple  $A = (a_1, a_2, \dots, a_n)$ .

*Definition 3.* Given iteration space  $\mathbb{I}$  and array space  $\mathbb{A}$ , each array access  $R_k$  in a loop nest is a function:  $R_k : \mathbb{I} \rightarrow \mathbb{A}$ .  $R_k$  is an affine array access if  $R_k(I) = M * I + V$ , where  $M$  is an  $n \times l$  matrix and  $V$  is a constant vector of size  $n$ .

*Example 1.*

```
for (i = 0; i <= N; i++)
  for (j = i; j <= M; j++) {
    A[i-1, i+j] = ...;
  }
```

For Example 1, the iteration space is

$$\mathbb{I} = \{(0, 0), (0, 1), \dots, (0, M), (1, 1), (1, 2), \dots, (1, M) \dots\}.$$

Assume the size of array  $A$  is  $10 \times 10$ , the data space of array  $A$  is

$$\mathbb{A} = \{(0, 0), (0, 1), \dots, (0, 10), (1, 0), (1, 1), \dots, (1, 10) \dots\}.$$

And array access  $R_1 : A[i-1, i+j]$  is an affine array access with respect to iteration space  $\mathbb{I} = (i, j)$  since

$$R_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} i-1 \\ i+j \end{bmatrix}.$$

If loop bounds and array accesses are affine, loop space and array accesses can be modeled as polytopes on rational space  $Q$ .

*Definition 4.* A rational parametric polytope  $PT_p$  with  $n$  parameters  $\vec{p}$  is a set of rational  $d$ -dimensional vectors  $\vec{x}$  in rational space  $Q^d$  defined by  $m$  linear inequalities on  $\vec{x}$  and  $\vec{p}$ , where  $A$  and  $B$  are  $(m \times d)$  and  $(m \times n)$  rational matrices, and  $c$  is a size- $m$  constant vector:

$$PT_p = \{\vec{x} \in Q^d \mid A\vec{x} \geq B\vec{p} + c\}.$$

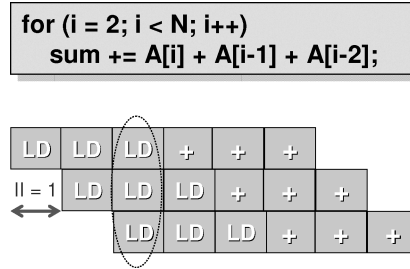


Fig. 2. A loop pipelining example.

We can easily construct the polytope which contains the iteration space from loop structures. For the example above, the iteration space is integer points inside polytope:

$$PT_I = \{(i, j) \in \mathcal{Q}^2 \mid 0 \leq i \leq N, i \leq j \leq M\}.$$

The corresponding values of  $A$ ,  $B$ ,  $\vec{p}$  and  $c$  in Definition 4 can also be calculated from the given form:

$$PT_I = \left\{ I \in \mathcal{Q}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ -N \\ 0 \\ -M \end{bmatrix} \right\}.$$

If array access  $R_k$  is an affine array access  $R_k(I) = M * I + V$ , the number of accesses on this element is the number of integer point inside the intersection of iteration space  $\mathbb{I}$  and the hyperplane  $\{I \mid A = M * I + V\}$ . For instance, the number of loop iterations that access array location  $A[a_1, a_2]$  is a polytope

$$\{(i, j) \mid i - 1 = a_1, i + j = a_2, 0 \leq i \leq N, i \leq j \leq M\}.$$

The polytope model provides a useful abstract of the array access behavior, which can be used to find the array access frequency for power optimization. Details will be discussed later.

## 2.2. Loop Pipelining

Loop pipelining is the most commonly used technique for improving throughput; it allows simultaneous execution of operations in several continuous loop iterations. Modulo scheduling [Rau 1994] is a loop pipelining technique which tries to find a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or different iterations. This constant interval between the start of successive iterations is termed the initiation interval (II), usually measured in clock cycles in synchronized hardware designs. An example is shown in Figure 2, which has  $II = 1$ .

*Definition 5.* The minimum initiation interval  $MII$  is a lower bound on the smallest possible value of II for which a modulo schedule exists.

The lower bound  $MII$  is limited by two factors: resource limits and inter-iteration loop dependence, which are termed as resource-constrained  $MII$  ( $ResMII$ ) and recurrence-constrained  $MII$  ( $RecMII$ ) respectively. Thus,  $MII \geq \text{MAX}(ResMII, RecMII)$ .

In a pipelined design, each operation in a loop body is executed once within  $II$  cycles, so obviously  $ResII * \text{resources} \geq \text{operations}$ , or  $ResII \geq \text{operations}/\text{resources}$ . For memory access, if the memory port number is  $N$ , and there are  $K$  memory accesses,

the final  $II$  is limited to  $II \geq K/N$ . Assume the underlying memory system has only 2 read ports, then  $ResII \geq 3/2 = 2$  for the case in Figure 2, which means  $II$  cannot be 1 without increasing memory bandwidth. This paper proposes an effective algorithm to improve  $II$  through memory partitioning.

Although resource constraints are the main focus of this article, loop dependence also imposes constraints on the final  $II$ . Our goal is to remove memory resource conflicts without affecting loop dependence constraints.

*Definition 6. Loop Dependence.* There exists a dependence from statement  $S1$  to statement  $S2$  if and only if there exist two iteration vectors  $i$  and  $j$  for the nest, such that (1)  $i \leq j$  and there is a path from  $S1$  to  $S2$  in the body of the loop, (2) statement  $S1$  on iteration  $i$  and statement  $S2$  on iteration  $j$  access the same memory location. In this case,  $j - i$  is called Dependence Distance  $D$ .

Loop dependences limit the freedom of operations in scheduling. If there is a dependence from  $S1$  to  $S2$  (scheduled at time step  $T(S1)$  and  $T(S2)$  respectively) with distance  $D$ , and the total combinational delay from operation  $S1$  to  $S2$  is  $delay(S1, S2)$ , we have the following constraints:

$$T(S2) + II * D - T(S1) \geq delay(S1, S2).$$

If there is a cycle  $c$  in the dependence graph, where each edge  $e$  represents a loop dependence between two operations, the lower bound of  $RecMII$  can be calculated based on the above equation:

$$RecMII \geq \frac{\sum_{e \in c} delay(e)}{\sum_{e \in c} distance(e)}.$$

In modulo scheduling, if array reference  $R_k$  is scheduled at time step  $t_k$  inside a loop body ( $t_k$  starts from 0), it will be executed together with any other array reference  $R_l$  when ( $t_k \equiv t_l \pmod{II}$ ). Assume that operation  $op$  is scheduled at time step 0 in the loop body (i.e.,  $t_{op} = 0$ ). If  $t_{op}$  is executed at loop iteration  $i$ ,  $R_k$  is executed at iteration  $i - \lfloor t_k/II \rfloor$ .

*Definition 7.* Offset  $\Delta I_k(t_k, II) = -\lfloor t_k/II \rfloor$  for an operation  $R_k$  is the difference between loop iterations of  $R_k$  and operation  $t_{op} = 0$ .

*Example 2.* In Figure 2, we have  $t_{A[i]} = 0$  and  $t_{A[i-1]} = 1$ . And  $\Delta I_{A[i-1]} = -t_{A[i-1]}/II = -1/1 = -1$ , which means if array access  $A[i]$  is executed at iteration  $i$ ,  $A[i-1]$  in iteration  $i-1$  is executed in the same clock cycle.

### 2.3. Memory Partitioning

Memory partitioning is a widely used techniques to improve memory bandwidth without data duplication. The original array will be placed into  $N$  nonoverlapping banks, and each bank is implemented with a separate memory block to allow simultaneous accesses on different banks. However, the original program must have inherited data parallelism to take advantage of memory partitioning.

*Definition 8.* Memory partitioning is described as a function  $P$  that maps array address  $A = (a_1, a_2, \dots, a_n)$  in array spaces to partitioned memory banks, that is,  $P(A)$  is the memory bank index that  $A$  belongs to after partitioning.

The two most commonly used data partition schemes are block partition and cyclic partition, as shown in Figure 3. These schemes are regular so that they can be easily implemented after partitioning; this is especially true for hardware synthesis since extra logic for addressing may increase the final area dramatically.

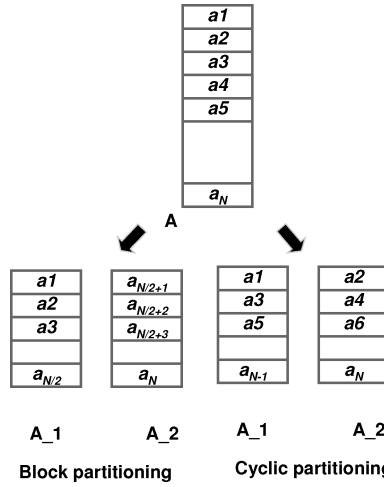


Fig. 3. Block and cyclic memory partitioning.

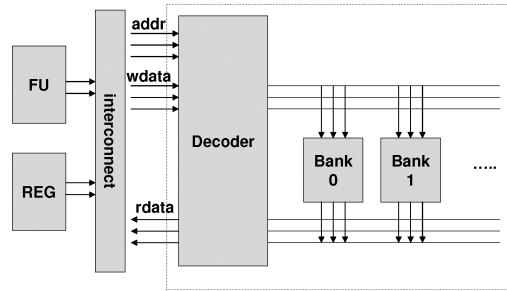


Fig. 4. Memory architecture.

*Example 3.* N-way 1-dimensional cyclic partitioning:

$$P(i) = i \bmod N.$$

*Example 4.* N-way 1-dimensional block partitioning:

$$P(i) = k \quad \text{if} \quad N \cdot k \leq i < (N + 1) \cdot k.$$

The physical implementation of a memory is shown in Figure 4. The original memory is partitioned into several banks, but the interface to function units (FUs) and registers (REGs) remains the same. Inside the partitioned memory, the decoding logic will dispatch memory access requests based on their addresses.

### 3. A MOTIVATIONAL EXAMPLE

The example being discussed in this section is taken from a recent work on FPGA acceleration [Cong and Zou 2008]. The goal of their research was to speed up a lithography simulation algorithm using behavioral synthesis tools. The kernel part of the lithography simulation program is a 2-level nest loop which updates a 2D image  $I$  based on a 2D kernel array  $K$ . To increase parallelism, the original loop structure is reorganized by loop tiling. The pseudocode with a  $2 \times 2$  tile size is shown in Figure 5.

Based on discussions in Section 2.2,  $II \geq \text{operations/resources}$  since all operations at least execute once in  $II$  clock cycles. For the lithography simulation problem, if we simply map each array into one on-chip memory on FPGA, port limitations determine

```

.....
//Core computation , 2X2 unrolling
for (x=0;x<L;x++)
  for (y=0;y<L;y++)
  {
    addr_x = 10*x-R[n]+c;
    addr_y = 10*y-R[n]+c;
    I[2x][2y] += (-1)n*K[addr_x][addr_y];
    I[2x+1][2y] += (-1)n*K[addr_x+5][addr_y];
    I[2x][2y+1] += (-1)n*K[addr_x][addr_y+5];
    I[2x+1][2y+1] += (-1)n*K[addr_x+5][addr_y+5];
  }
.....

```

Fig. 5. Pseudocode of lithography simulation.

2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1	0	1	0	1	0	1

Fig. 6.  $2 \times 2$  partitioning and 4 array accesses on array  $K$ .

that the best  $II$  achievable is 4, assuming that memories on the target platform only have one read port and one write port.

One simple solution is to perform a  $2 \times 2$  cyclic partitioning on both arrays  $I$  and  $K$ . The numbers 0, 1, 2, 3 denote the four banks after partitioning. It is clear that the four array accesses in the same iteration in Figure 5 on array  $K$  will always be distributed to different banks as shown in Figure 6. As a result, the final  $II$  can be reduced to 1 after partitioning.

For synchronous hardware design, memory partitioning must be considered together with scheduling to ensure that the final solution port limitation is not violated at the cycle-accurate level. For a better illustration, a reduced version of the lithography simulation program is analyzed:

*Example 5.*

```

for (i =0; i <= N; i++) {
... = K[i] + K[i + 5];
}

```

If these two array accesses are scheduled at the same cycle for a 2-way cyclic partitioning, they will always access different memory banks, as shown in Figure 7(a) (the dotted rectangle is the loop body, the element with lighter color is accessed by  $K[i]$ , and the darker color by  $K[i + 5]$ ; numbers 0, 1 denote the two banks after partitioning). However, if we schedule the data access  $K[i + 5]$  one cycle after data access  $K[i]$ , after certain steps, they will access the same memory bank under loop pipelining with  $II = 1$ , as shown in Figure 7(b). Thus only memory partitioning is not enough for hardware design, since their access pattern may be quite different depending on the scheduling results. An important contribution of our approach is that our memory partitioning

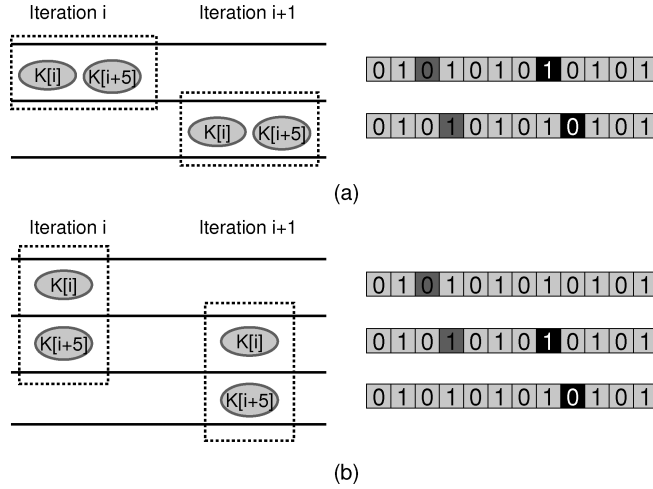


Fig. 7. Impact of scheduling on memory partitioning.

approach is combined with memory access scheduling in a cycle-accurate way unlike previous approaches.

#### 4. PROBLEM FORMULATION

To overcome the memory bandwidth problem, an automatic memory partitioning algorithm is developed in this paper. In our first formulation, cycle-accurate scheduling information is not considered.

*Problem 1.* Given a computation kernel specification in a  $l$ -level loop nest,  $K$  array references  $R_k$  on the same array  $A$ , target throughput requirement  $II$ , memory port limitation  $N$ , and a platform-dependent partitioning cost function  $cost(P)$ , the goal is to find a min-cost partition  $P$  such that:

$$\forall j \max(|S_j|) \leq II * N, \quad S_j = \{R_k \mid \exists I \in \mathbb{I}, P(R_k(I)) = j\} \quad (1)$$

Cost function  $cost(P)$  is platform dependent, and for most hardware designs, the cost of a partitioning is proportional to the number of memory banks generated. Set  $S_j$  contains all the array references which may access bank  $j$  after partitioning. And Equation 1 ensures that all possible array accesses on memory bank  $j$  are less than the required value so that final design can potentially meet final throughput constraints  $II$ .

*Example 6.*

```
for (i = 0; i <= N; i++)
  t += A[2*i] + A[2*i+1];
```

If  $P(i) = i \bmod 2$ , we have  $S_0 = \{A[2*i]\}$  and  $S_1 = \{A[2*i+1]\}$ . And  $II$  will be reduced from 2 to 1 on a single read-port memory design.

However, cycle-accurate information needs to be considered for more general cases. For the example in Figure 7, any of array accesses  $K[i]$  and  $K[i+5]$  can access either bank 0 or 1. Therefore memory bottlenecks cannot be eliminated by solving Problem 1. If they are scheduled at the same time step as Figure 7(a), these two accesses will never access the same memory bank, and  $II$  can be reduced to 1 for this case; but  $II$  cannot be improved if they are scheduled as shown in Figure 7(b).

Based on this observation, a throughput optimization problem which considers cycle-accurate memory access patterns is formulated in Problem 2.

*Problem 2.* Given a computation kernel specification in a  $l$ -level loop nest,  $K$  array references  $R_k$  on the same array  $A$ , target throughput requirement  $II$ , memory port limitation  $N$ , and a platform-dependent partitioning cost function  $cost(P)$ , the goal is to find a min-cost partition  $P$  as well as scheduling steps  $t_k$  for each  $R_k$ , such that:

$$\forall I \in \mathbb{I} \max(|S_{j,t}(I)|) \leq N \quad \text{where} \\ S_{j,t}(I) = \{R_k | (P(R_k(I + \Delta I_k)) = j) \text{ and } (t_k \bmod II = t)\}.$$

$\Delta I_k$  is defined in Section 2.2, since different operations may execute at different loop iterations. All the memory accesses can be divided into  $II$  groups, where group  $t$  contains all memory accesses  $t_k \bmod II = t$ . Intuitively, set  $S_{j,t}$  is the set of all the array accesses which may access memory bank  $j$  for group  $t$ . The above equation checks a fine-grained condition at cycle level to make sure port limitation is not violated. Overall, the memory partitioning problem attempts to solve the necessary condition for loop pipelining; the final throughput result  $II$  is not guaranteed to be achieved when data dependences are considered.

*Example 7.* Let  $R_1 = K[i]$  and  $R_2 = K[i + 5]$  for array accesses in Figure 7(a), and target  $II = 1$ , it is easy to see that  $\Delta I_1 = \Delta I_2 = 0$ . And for cyclic partition  $P(i) = i \bmod 2$ ,

$$S_{0,0}(i) = \begin{cases} \{K[i]\} & \text{if } i \bmod 2 = 0 \\ \{K[i + 5]\} & \text{otherwise} \end{cases} \\ S_{1,0}(i) = \begin{cases} \{K[i + 5]\} & \text{if } i \bmod 2 = 0 \\ \{K[i]\} & \text{otherwise.} \end{cases}$$

Overall,  $\forall i \quad \max(|S_{j,t}(i)|) \leq 1$ , and  $II$  is reduced to 1.

Similarly, we can also prove that  $\max(|S_{j,t}(i)|) = 2$  for the scheduling in Figure 7(b), which means  $II = 2$  under the same cyclic partition. Therefore, the memory partitioning problem must be considered together with memory access scheduling to achieve the optimal throughput.

Power optimization has emerged as a first-line optimization objective for modern technology. Our proposed algorithm targets power optimization as well after throughput constraints are met. After power-aware memory partitioning, only those memory banks which are accessed in the current time step will be activated by the clock-enable signals. Therefore, inactive banks can be put in “standby” or “sleep” mode, and consume nearly zero dynamic power. So the complete problem for both throughput and power optimization is formulated as the following:

*Problem 3.* Given a computation kernel specification in a  $l$ -level loop nest,  $K$  array references  $R_k$  on the same array  $A$ , target throughput requirement  $II$ , memory port limitation  $N$ , and a platform-dependent energy consumption function  $C(P)$ , the goal is to find a partition  $P$  such that throughput constraints are met and overall energy is minimized.

Problem 3 is formulated to solve the partitioning problem in one loop nest. If an array appears in multiple loop nests, the final partition solution can be easily constructed by combining the partitioning results from each loop nest. And we assume that each array will be assigned to different memory blocks so that they will never conflict to

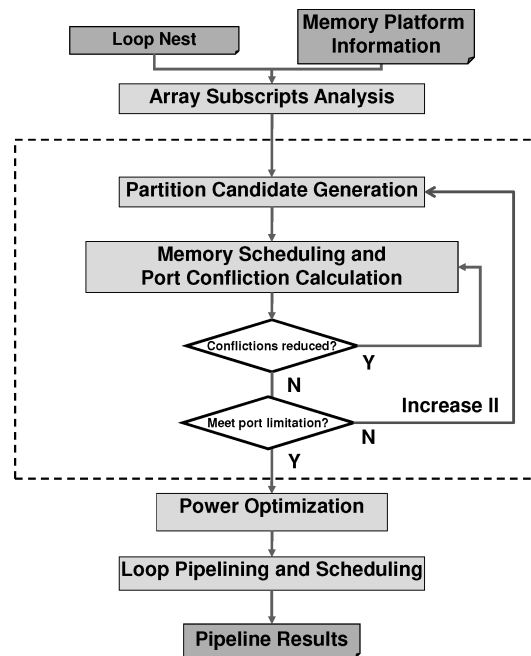


Fig. 8. Memory partitioning algorithm overview.

each other. Therefore, our algorithm can process each array separately to handle loops with multiple arrays.

## 5. PARTITIONING ALGORITHM

In the next three sections, a novel algorithm is presented to effectively solve the memory partitioning problem together with the scheduling problem. The goal of throughput optimization is to maximize concurrent memory accesses by partitioning, however power optimization tries to shut down unused memory banks to save dynamic power consumption. These distinct objectives make it extremely difficult to solve Problem 3 in one step. In our approach, the throughput optimization is solved at the first step, then the results are passed to the power optimization step to reduce the total energy consumption. The overview of our algorithm is shown in Figure 8.

The original application specifications are parsed in and optimized by compilers. Computation kernels for speedup are identified by profiling and provided as the input to the memory partitioning algorithm. Advanced loop optimization techniques (such as loop tiling, loop interchange, memory reuse, etc.) can be deployed in a preprocessing pass to further remove redundant memory accesses and increase data parallelism. Based on target throughput, memory accesses that violate the port limitation are analyzed and grouped for partitioning. The first step of our proposed algorithm is to find good partition candidates based on array references and memory platform information; after candidates are generated, a branch and bound algorithm will search the best combination of partitions on different array dimensions to meet the port limitation and minimize the partition cost. If no partition solution can be obtained with the current  $II$ ,  $II$  will be increased until port limitation is met. After throughput constraints are met, partitioning results will be passed to the power optimization step to minimize power consumption.

## 6. THROUGHPUT OPTIMIZATION

### 6.1. Partition Candidates Generation

The two most commonly used data partition schemes are block partition and cyclic partition. Although any partition scheme can be integrated into our flow, only *cyclic* partitioning is discussed in the current implementation. The seemingly restricted partitioning form actually does not limit the effectiveness of our algorithm very much because the majority of array accesses in loop kernels are simple and regular based on the research in Kennedy and Allen [2002], which shows that the simple cases (zero induction variable, single induction variable) account for about 80% of overall array accesses. This is especially true for applications in the multimedia domain since these program tend to access adjacent array elements and have strong space locality. Our experiments also verify our intuition.

As we have mentioned, we assume that all array accesses are affine in this section. Our discussion starts from a simple case: two array accesses  $R_1$  and  $R_2$  on a 1-dimensional array in a 1-level loop nest. Assume that the induction variable is  $i$ , array accesses can be represented by  $R_1 = a_1 * i + b_1$  and  $R_2 = a_2 * i + b_2$ . A good partition should guarantee that  $R_1$  and  $R_2$  never access the same memory bank after partitioning. If  $a_1 = a_2$ , it is trivial to see that if  $(b_1 - b_2) \bmod N \neq 0$ , these two array accesses will never access the same memory bank. If  $a_1 \neq a_2$ , the following theorem provides a criteria for selecting partition factor  $N$ .

THEOREM 1.

$$\begin{aligned} & \forall i \quad a_1 \cdot i + b_1 \neq a_2 \cdot i + b_2 \pmod{N} \\ \Leftrightarrow & \gcd(a_1 - a_2, N) \nmid (b_2 - b_1) \end{aligned}$$

PROOF.

$$\begin{aligned} & \exists i \text{ s.t. } a_1 \cdot i + b_1 \equiv a_2 \cdot i + b_2 \pmod{N} \\ \Leftrightarrow & \exists i, k \text{ s.t. } (a_1 - a_2) \cdot i + k \cdot N = b_2 - b_1 \\ \Leftrightarrow & \gcd(a_1 - a_2, N) \mid (b_2 - b_1) \end{aligned}$$

The converse-negative proposition of the original theorem has been proved.  $\square$

*Example 8.* Consider array accesses  $A[i]$  and  $A[3 * i + 1]$ , the best cyclic partition factor is 2, since  $\gcd(3 - 1, 2) \nmid (1 - 0)$ .

It is easy to extend Theorem 1 to handle  $K$  array accesses in a  $l$ -level loop nest. Based on Theorem 1, integer values are examined to find the best candidates which balance array accesses to different memory banks. Please notice that factors which are power of 2 always have a higher priority than other factors since the modulo operation can be transformed to shift operation for the ease of hardware implementation.

### 6.2. Throughput Optimization

After partitioning candidates set  $C$  on every array dimension is generated, the memory partitioning algorithm starts a branch and bound search process to minimize concurrent array access at each clock cycle and finalize the partitioning solution. The simplified version in Problem 1 is solved first. If it can be solved, then the cycle-accurate formulation does not need to be considered. This may reduce the search space dramatically on many of the test cases. After partitioning candidates have been generated, the partitioning algorithm attempts to solve a subproblem of Problem 2.

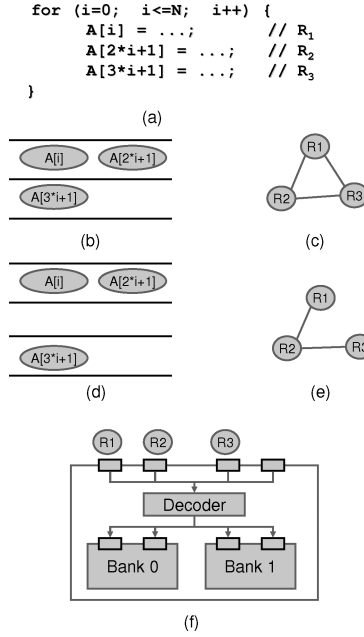


Fig. 9. An illustration example.

**Problem 4.** Given  $K$  array references  $R_k$  on the same array  $A$ , the target throughput requirement  $II$ , and the partition  $P$ , find a scheduling step  $t_k$  for each  $R_k$ , such that:

$$\text{minimizes: } \max(|S_{j,t}(I)|) \quad \forall I \in \mathbb{I}$$

$$S_{j,t}(I) = \{R_k | (P(R_k(I + \Delta I_k)) = j) \text{ and } (t_k \bmod II = t)\}$$

This problem is similar to Problem 2 except that the partitioning solution is given. However, it is very difficult to solve this problem because of the universal condition on the iteration space. The essential goal of this problem is to minimize concurrent array accesses on each memory bank after partitioning. Therefore, we try to solve a relaxed version of Problem 4 by introducing the concept of conflict graph.

**Definition 9.** Let  $\text{conflict}(op_1, op_2, i_1, i_2)$  be the function for checking whether  $op_1$  scheduled at time step  $i_1$  may conflict with  $op_2$  scheduled at  $i_2$  in certain loop iteration, that is,  $\text{conflict}(op_1, op_2, i_1, i_2) =$

$$\begin{cases} 1 & \text{if } \exists I \{(P(R_i(I + \Delta I_i)) = P(R_j(I + \Delta I_j))) \text{ and } (t_i \equiv t_j \bmod II)\} \\ 0 & \text{otherwise.} \end{cases}$$

**Definition 10.** Given  $K$  array references  $R_k$  on the same array  $A$ , and scheduling step  $t_k$  for each  $R_k$ , the conflict graph  $G = (V, E)$  is a undirected graph where vertices in  $V$  correspond to array accesses, and edge  $(v_i, v_j) \in G$  if

$$\text{conflict}(op_i, op_j, t_i, t_j) = 1.$$

The conflict graph captures pairwise conflict information between two array accesses. For any given cyclic partition, it is very easy to construct conflict graphs with the *gcd* test technique in Section 6.1. The conflict graph is more conservative since if two array accesses conflict in *any* iteration, these two operations will be considered conflicting at *every* loop iteration. The conflict graph depends on scheduling results.

**Example 9.** The loop in Figure 9(a) is initially scheduled as Figure 9(b), and the corresponding conflict graph is shown in Figure 9(c) assuming target  $II = 1$ . There is

an edge between  $R_1$  and  $R_3$ , because  $\Delta I_{R_1} = 0$  and  $\Delta I_{R_3} = -1$  and  $\forall i \ i \equiv (3 * (i - 1) + 1) \pmod{2}$ . The other edges of this conflict graph can be inferred in a similar way.

**THEOREM 2.** *Given  $K$  array references  $R_k$  on the same array  $A$ , scheduling step  $t_k$  for each  $R_k$ , and  $G$  is the corresponding conflict graph:*

$$\max(|S_{j,t}(I)|) \leq \text{maximum\_clique}(G) \quad \forall I \in \mathbb{I}.$$

**PROOF.** It is easy to see that for any two nodes  $i$  and  $j$  in set  $S_{j,t}(I)$ , there is an edge in the corresponding conflict graph  $G$  since at iteration  $I$  these array accesses  $R_i$  and  $R_j$  visit the same memory bank. As a result,  $S_{j,t}(I)$  corresponds to a clique in  $G$ , and its size will certainly be less than or equal to the maximum clique of  $G$ .  $\square$

The maximum clique of  $G$  provides an upper bound for  $S_{j,t}(I)$ , and a very tight one in general. Based on Theorem 2, our proposed algorithm solves the following relaxed problem of Problem 4.

**Problem 5.** Given  $K$  array references  $R_k$  on the same array  $A$ , the target throughput requirement  $II$ , and the partition  $P$ , find a scheduling  $t_k$  for each  $R_k$ , such that the maximum clique of the corresponding conflict graph  $G$  is minimized.

Despite the NP-completeness of the maximum clique problem, recent research has shown that exact methods can find maximum cliques in graphs of over 400 nodes [Bomze et al. 1999]. Fortunately for our problem, the number of array accesses is under that scale.

**6.2.1. ILP Algorithm.** Firstly, an ILP algorithm is proposed to solve Problem 5 optimally for a given latency constraint  $L$ , that is, the latency of the loop body of the perfect loop nest is  $\leq L$ .  $L$  can be obtained by a previous scheduling step in an iterative flow, or by an estimate step based on resource constraints.

In our implementation, the maximum clique problem is solved with integer linear programming (ILP) by transforming it to the maximum independent set problem on the complement graph  $\bar{G}$  of  $G$ .

The ILP formulation consists of a set of Boolean variables. Variable  $x_{ik} = 1$  if memory operation  $op_i$  is scheduled at time step  $k$ . Variable  $e_{ij} = 1$  means edge  $\langle i, j \rangle$  exists in graph  $\bar{G}$ . Variable  $n_i = 1$  if operation  $op_i$  is selected in a maximum independent set of graph  $\bar{G}$ .

We introduce the following constraints:

—if  $\text{conflict}(i, j, m, n) = 1$ , then  $e_{ij} = 0$ .

$$x_{im} + x_{jn} + e_{ij} \leq 2. \quad (2)$$

—if  $\text{conflict}(i, j, m, n) = 0$ , then  $e_{ij} = 1$ .

$$x_{im} + x_{jn} - e_{ij} \leq 1. \quad (3)$$

—each operation can only be scheduled at one time step.

$$\sum_{k=1,L} x_{ik} = 1. \quad (4)$$

—if there is an edge  $\langle i, j \rangle$ , then  $n_i$  and  $n_j$  cannot be both selected to an independent set.

$$n_i + n_j + e_{ij} \leq 2. \quad (5)$$

Objective function is to maximize operations selected into independent set, that is,

$$\max \sum_{i=1,M} n_i. \quad (6)$$

We also consider recurrence of dependence such that our scheduling will not increase  $II$  in addition to resource constraint.

As discussed in Section 2.2, if there is a dependence from  $t_i$  to  $t_j$  with distance  $D$ , we have the following constraints:

$$t_i + II * D - t_j \geq \text{delay}(i, j).$$

Translating this constraint into our formulation, we have

$$x_{im} + x_{jn} \leq 1 \text{ if } (m + D * II - n < \text{delay}(i, j)), \quad (7)$$

which means  $op_i$  and  $op_j$  can not be scheduled at time step  $m$  and  $n$  if they don't satisfy the recurrence constraint.  $\text{Delay}(i, j)$  is the minimum delay from operation  $op_i$  to  $op_j$ . Since we don't allow chaining between memory accesses, we can assume  $\text{delay}(i, j) = 1$  for most memory accesses.

Assume there are  $M$  memory operations to be scheduled, the number of variables is  $ML + M^2 + M = O(ML + M^2)$ , and the number of constraints is  $O(M^2L^2)$ . It works well since  $M$  and  $L$  are not very big in practice.

---

**ALGORITHM 1:** Search Algorithm
 

---

```

1:  $\mathbb{C}_d$       → partition candidates in array dimension  $d$ 
2:  $\mathbb{R} = \{R_1, R_2, \dots, R_K\}$  → array accesses
3:  $\mathbb{S}$       → set of partition candidates on different dimensions
4:  $\mathbb{T}$       → scheduling results
5:  $N$       → memory port limitation
6:
7:  $\mathbb{T} = \text{get\_initial\_schedule\_solution}()$ 
8: for array dimension  $d$  do
9:   for all  $c_i \in \mathbb{C}_d$  do
10:    update current partition solution  $\mathbb{S}'$  by combining  $c_i$ 
11:    update_conflict_graph( $G, \mathbb{R}, \mathbb{T}, \mathbb{S}'$ )
12:     $MC = \text{maximum\_clique}(G)$ 
13:    while  $|MC| > N$  and  $|MC|$  decreases do
14:       $\mathbb{T} = \text{reschedule\_nodes}(MC)$ 
15:      update_conflict_graph( $G, \mathbb{R}, \mathbb{T}, \mathbb{S}'$ )
16:       $MC = \text{maximum\_clique}(G)$ 
17:    end while
18:    compare cost of current solution  $\text{cost}(\mathbb{S}')$  with  $\text{cost}(\mathbb{S})$ 
19:    update  $\mathbb{S}$ 
20:   end for
21: end for
22: return  $\mathbb{S}$ 

```

---

**6.2.2. A Heuristic Algorithm.** The aforementioned ILP formulation could have a scalability problem in bigger designs. A heuristic algorithm is proposed in this section which is almost as effective as the ILP formulation with much better scalability. Our experiment shows that our proposed heuristic algorithm achieve exactly the same results with the ILP algorithm on a set of multimedia programs.

The pseudocode of our algorithm for Problem 5 is shown in Algorithm 1. For a given set of partition candidates, an initial schedule  $\mathbb{T}$  is obtained from a previous scheduling stage. Each partition candidate  $c_i$  on one specific array dimension is combined with candidates on other array dimensions to form the current solution  $\mathbb{S}'$ . With scheduling information and  $\mathbb{S}'$ , we can get the corresponding conflict graph  $G$  and maximum clique ( $MC$ ). If  $|MC|$  meets the port limitation  $N$ , the solution is returned. Otherwise, a rescheduling step will attempt to reduce  $|MC|$  as shown in line 14.

The rescheduling step then reschedules nodes in  $MC$ ; each node will be tentatively moved to later time steps until it will not conflict with at least one of other nodes in  $MC$ . The edge number change in  $G$  is also tracked for each node, and the node which reduces most edges is selected by the rescheduling pass and its move is committed.

This process iterates until  $|MC|$  is minimized. If  $|MC| \leq N$ , we get a feasible partition solution, otherwise further partitioning is applied on other array dimensions. The final min-cost solution is selected from all feasible solutions, as shown in line 18.

*Example 10.* Let  $N = 2$  for the example in Figure 9. Figure 9(b) is the initial scheduling, and  $|MC| = 3 > N$  for the conflict graph in Figure 9(c). A rescheduling result is shown in Figure 9(d), and the corresponding conflict graph is in Figure 9(e). There is no edge between  $R_1$  and  $R_3$ , because  $\Delta I_{R_1} = 0$  and  $\Delta I_{R_3} = -2$ , and  $\forall i, i \neq (3 * (i - 2) + 1) \bmod 2$ . After rescheduling,  $|MC| = 2 \leq N$  and final  $II = 1$ . The final memory implementation and port binding are shown in Figure 9(f); these 3 memory accesses can execute at the same clock cycle since at most 2 of them can access the same bank.

**THEOREM 3.** *The time complexity of Algorithm 1 is  $O(C \cdot D \cdot K^3 \cdot L)$ , where  $D$  is the array dimension,  $C$  is the maximum number of partition candidates on any array dimension,  $K$  is the number of array accesses in a loop nest, and  $L$  is the latency of the loop nest.*

**PROOF.** It is obvious that  $|MC| < K$  since the maximal clique of a graph at most equals to the size of the graph. And the complexity of reschedule step is  $O(K^2 \cdot L)$ , because each of the  $K$  array accesses is moved by at most  $L$  steps, and at each step  $K$  computations are needed to update the conflict graph. So the complexity of the innermost loop is  $O(K^3 \cdot L)$ , and overall the complexity is  $O(C \cdot D \cdot K^3 \cdot L)$ .  $\square$

### 6.3. Pipelining Algorithm Enhancement

Memory partitioning removes obstacles in memory port limitations for potential throughput improvement and improves the lower bound  $MII$ ; however it does not guarantee that the target  $II$  will be achieved in the loop pipelining and scheduling step because of data dependence. Even though the memory partitioning solution is valid, a bad scheduling may still impair the final performance. So, an enhanced pipelining algorithm is proposed under the direction of the memory partitioning algorithm to achieve better throughput.

Modulo scheduling is a technique for software pipelining that schedules one iteration of the loop in such a way that successive iterations can repeat the same schedule at constant intervals without violating dependency constraints and resource constraints [Allan et al. 1995]. Typical modulo scheduling algorithms like iterative modulo scheduling in [Rau 1994], use a modulo resource reservation table to keep track of the resource usages of already scheduled operations. When scheduling a new operation in a certain control step, the reservation table is checked for potential resource hazards. The traditional reservation table is too conservative to handle this since memory references may access all memory banks in different iterations; although as a group, no port limitation will be violated following our memory partitioning algorithm. For the example in Figure 9, array accesses  $A[i]$ ,  $A[2 * i + 1]$  and  $A[3 * i + 1]$  all access bank 1 in different loop iterations. Therefore, all three array accesses reserve bank 1 using the traditional reservation table. We can not achieve  $II = 1$  because bank 1 only has two ports.

The modulo scheduling algorithm can be improved based on the conflict graph. For example, any of  $A[i]$  and  $A[3i + 1]$  may access banks 0 and 1. However, based on *conflict* function in Definition 9, we know that they will never access the same bank at the same cycle if they are scheduled at the same clock cycle, as a result they can be assigned to the same port of the partitioned array memory  $A'$ . With this modification, high-quality

schedules can be generated using typical modulo scheduling algorithms such as the iterative modulo scheduling [Rau 1994]. According to Rau [1994], about 96% of designs can achieve  $II = MII$  which means throughput will be improved on most design with our techniques. This is further confirmed by our experimental results.

#### 6.4. Handle Irregular Array Access

Our approach can be extended to handle certain irregular array accesses like indirect access in the lithography simulation. Array subscripts are analyzed recursively by compilers. The basic element in the expression tree is either an induction for affine array accesses or irregular access such as non-affine expression ( $i * j$ ) and indirect array access ( $A[B[i]]$ ). In our implementation, the LLVM compiler is used as the compiler which can effectively analyze expressions in its “ScalarEvolution” pass [<http://llvm.cs.uiuc.edu>]. To handle irregular array subscripts, each irregular element in the expression tree is added to the iteration space as a pseudoinduction variable. These pseudo induction variables are different than normal induction variables because you can never know how they change against loop iterations. Therefore, if two irregular array accesses execute at different loop iterations, it is very difficult to tell whether they will access the same memory bank or not after partitioning.

*Example 11.* For the lithography simulation program in Figure 5, irregular array access address  $addr_x = 10 * x - R[n] + c$  ( $c$  is a constant) has two basic elements  $x$  and  $R[n]$ . The original iteration space  $I = \{x, y\}$  is changed to  $I' = \{x, y, z\}$  where  $z = R[n]$ , such that  $addr_x = 10 * x + z + c$  is affine with respect to  $I'$ .

With extended iteration space  $I'$ , the conflict graph  $G$  will be constructed differently to reflect the changes.

*Definition 11.* Given  $K$  irregular array references  $R_k$  on the same array  $A$ , and scheduling step  $t_k$  for each  $R_k$ , the extended conflict graph  $G' = (V, E)$  is an undirected graph where vertices in  $V$  correspond to array accesses, and edge  $(v_i, v_j) \in G$  if:

$$\Delta I_i \neq \Delta I_j \text{ or } \exists I (P(R_i(I)) = P(R_j(I))) \text{ and } (t_i \equiv t_j \text{ mod } II)$$

In this case, the conflict condition is even more restrictive for irregular array accesses. If two array accesses are executed at different iterations, they are always assumed to conflict with each other. With the modified conflict graph, our search algorithm can be easily updated to handle certain irregular array accesses. It is effective for many practical designs if array accesses can be grouped together and execute at the same loop iteration after pipelining.

## 7. POWER OPTIMIZATION

To meet the increasing demands for low power consumption in SoC design, a power optimization algorithm is proposed to reduce dynamic power consumption after meeting throughput constraints. Each memory bank generated by the previous throughput optimization algorithm will be further partitioned to reduce power, and throughput will not be adversely impacted by the power optimization step. The power optimization can be used to reduce power of the original design as well without performing the throughput optimization. Our approach uses a similar approach Poncino et al. [2000]. The original memory is partitioned into several continuous segments, where each segment is stored in a separate bank. Each memory bank is in “standby” mode to save power when none of its contents is accessed. The goal is to group frequently visited elements into small memory banks, because average power consumption decreases with memory

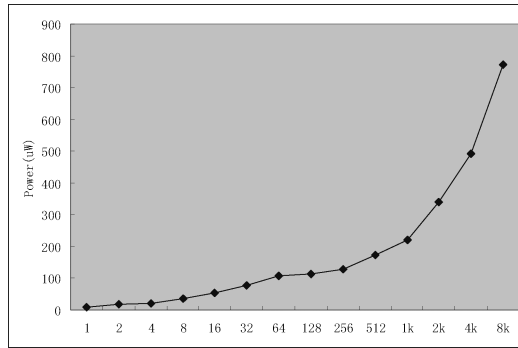


Fig. 10. Power consumption of a memory read access against the memory size.

size. The algorithm in this section solves the following problem, which is a refinement of Problem 3 with platform information.

*Problem 6.* Given a computation kernel specification in a  $l$ -level loop nest,  $K$  array references  $R_k$  on the same array  $A$ , and a platform-dependent energy consumption function  $E(s)$  on memory bank size  $s$  and a cost function of adding memory banks  $\Delta(n)$ , the goal is to find a partition  $P$  such that overall energy is minimized.

Power estimation and modeling has been well studied in past literature. Briefly, power modeling can be divided into two categories: black box approaches [Gupta and Najm 1997] and white box approaches [Landman and Rabaey 1995]. Black box approaches ignore internal implementation of memory design, and build power model from statistical results of simulations. White box approaches model the internal structures in an analytical method based on design parameters, such as switch activities, capacitance and voltage, etc. In our approach,  $E(s)$  is model by a black-box approach, where a set of sample power data are collected by simulation. The power data are reported by Magma Talus tool, and only dynamic power is considered in our problem. Power data between sample points are calculated by linear interpolation. Our power model does not consider the order of memory accesses since the memory access sequence is preserved; the only change is inside each memory blocks where one single block is partitioned into multiple blocks. We also use clock gating instead of power gating in our experiment. An example of the memory power-size trade off curve in the TSMC 90nm library<sup>2</sup> is shown in Figure 10.

Our algorithm is developed in a way similar to the work in Poncino et al. [2000]. Compared with Poncino et al. [2000], our approach has the following advantages: (a) The array access profile can be statically calculated for affine array accesses, or it can be represented by a function of unknown parameters in the program. Therefore, the tedious work of profiling can be greatly reduced for faster design space exploration. (b) Our approach can support both the “block” partitioning and “cyclic” partitioning to take advantage of access patterns of the original program.

### 7.1. Array Access Profile

The array access profile gives detailed access information for each array element. The essential part of the array access profile is to calculate  $f(A)$ , which is the number of array accesses on the array element  $A$ . To illustrate our techniques, a reduced version of the example in Section 2.1 is shown in the following.

<sup>2</sup><http://www.tsmc.com>.

*Example 12.*

```

for (i = 0; i <= N; i++)
  for (j = i; j <= M; j++)
    t += A[i + j];

```

Iteration spaces and array spaces can be modeled as integer points in rational polytopes as discussed in Section 2.1. For the example above, the number of loop iterations which load  $A[a]$  equals the number of integer point inside polytope:

$$PT_a = \{\langle i, j \rangle \in \mathbb{Q}^2 \mid 0 \leq i \leq N, i \leq j \leq M, i + j = a\}$$

The corresponding values of  $A$ ,  $B$ ,  $\vec{p}$  and  $c$  in Definition 4 can also be calculated from the previous form;

$$\left\{ \langle i, j \rangle \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} * [a] + \begin{bmatrix} 0 \\ -N \\ 0 \\ -M \\ 0 \\ 0 \end{bmatrix} \right\}$$

It's straightforward to  $f(a)$  equals the number of integer point in polytope  $PT_a$ . Counting integer point inside a polytope is a fundamental problem in mathematics, and has wide application in program transformations and optimizations. The *barvinok* library [Verdoolaege et al. 2007] can solve this problem in polynomial time, and the solution is a quasi-polynomial.

*Definition 12.* A rational  $n$ -periodic number  $U(\vec{p})$  is a function  $\mathbb{Z}^n \rightarrow \mathbb{Q}$ , such that there exists a period  $\vec{q} = (q_1, \dots, q_n)$ :

$$U(\vec{p}) = U(\vec{p}') \text{ when } (\forall i \ p_i \equiv p'_i \pmod{q_i})$$

*Definition 13.* A *quasi-polynomial* is a polynomial with periodic numbers as coefficients.

A periodic number is often represented by the fracture function  $\{x\} = x - \lfloor x \rfloor$ , where  $\lfloor x \rfloor$  is the integer part of a rational number.

*Example 13.*  $\{N/2\}$  is a 1-periodic number (0, 0.5) with a period 2, and  $\{N/2\} \cdot N + 1$  is a quasi-polynomial.

The number of array accesses on  $A[a_1]$  is calculated by the *barvinok* library, and the final result is a list of quasi-polynomials on a set of data domains:

$$\begin{cases} a_1/2 - \{a_1/2\} + 1, & 2N - a_1 \geq 1, M - a_1 \geq 1 \\ M - a_1/2 - \{a_1/2\} - 1, & 2N - a_1 \geq 1, M \leq a_1 \leq 2M \\ N + 1, & a_1 \geq 2N, M \geq a_1 + 1 \\ N + M - a_1 + 1, & M \geq a_1 \geq 2N, N + M \geq a_1 \end{cases}$$

If  $N$  and  $M$  are constants, we can directly calculate access frequency for any position  $A[a_1]$  from the previous equation. For instance, if  $N = M = 100$  and  $a \leq 5$ , the number of accesses on array element  $A[a_1]$  should be  $a_1/2 - \{a_1/2\} + 1 = \lfloor a_1/2 \rfloor + 1$  based on Equation (8). The result is illustrated by a graph in Figure 11. All iterations  $\langle i, j \rangle$  which access the same memory location are on the same diagonal line, and it is clear to see that  $f(a_1) = \lfloor a_1/2 \rfloor + 1$ .

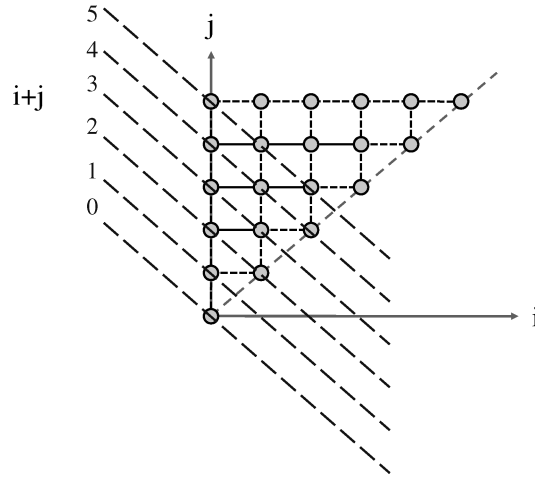


Fig. 11. Example of counting integer points inside a polytope.

If  $N$  and  $M$  are unknown, users only need to provide information on these variables. Compared to Poncino et al. [2000], our approach requires only small efforts from users to obtain an array access profile, which is a huge advantage to improve the design productivity and broaden design space exploration. Our approach is especially useful for multimedia programs or DSP designs, where most of array accesses are affine and no profiling is needed subsequently.

## 7.2. Search

After array profile  $f(A)$  is obtained, an iterative search step will find partitioning solutions based on the platform information. Overall, our approach uses an approach similar to Poncino et al. [2000] to search for the optimal solution of Equation (8). The optimization objective is formulated as the following:

$$\sum_{i=1}^n \left( \sum_{P(A)=i} f(A) \cdot E(S(i)) + \Delta(i) \right). \quad (8)$$

$E(s)$  is the energy consumption of accessing one element in a size  $s$  memory bank;  $S(i)$  is the size of bank  $i$  after partitioning;  $f(A)$  is the number of array accesses on the array element  $A$ ; and  $\Delta(n)$  is the cost of adding  $n$ th bank into an existing memory. Overall, the equation above represents the estimated energy consumption for a given partitioning solution  $P$ .

The algorithm in Poncino et al. [2000] is a recursive algorithm based on two assumptions:

- the total energy consumption of a memory bank monotonically increases with bank size;
- the number of memory banks  $n$  is much smaller than the total memory size  $m$ .

The basic idea of the algorithm in [Poncino et al. 2000] is to find a size  $n - 1$  cut set which separate the original memory with size  $m$  into  $n$  continuous segments. Clearly, the complexity of the algorithm grows with the number of combinations taking  $n - 1$  things from  $m$  things, or  $C_{n-1}^m$ . However, the minimal energy solution is updated to bound the searching process during the recursion, and the algorithm in Poncino et al. [2000] performs very well in practice.

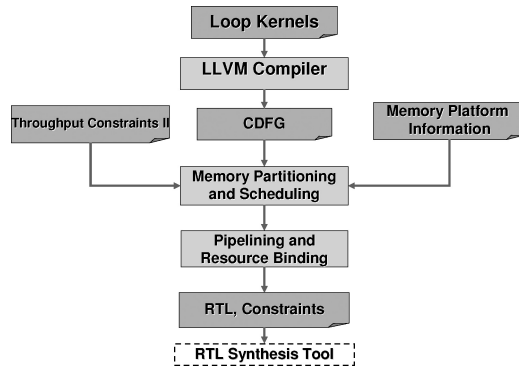


Fig. 12. Implementation flow.

The optimal solution is limited to one specific SRAM architecture called “segment configuration” in Poncino et al. [2000]. In this mode, a continuous segment of the original memory in range  $(lo, hi)$  is put in a separate memory bank. In contrast, our SRAM model can support more partitioning schemes, including *cyclic* partitioning. The benefit of cyclic partitioning can be illustrated by the following example:

*Example 14.* for  $(i = 0; i \leq N; i++)$   
 $t += A[2 * i] + A[2 * i + 1];$

It is clear that those two array accesses will be split to banks 0 and 1 respectively for partitioning  $P(a) = a \bmod 2$ . Therefore, no decoder is needed compared to the “segment configuration” architecture, while only one bank is activated for each original array access. If a clean partitioning is obtained,  $\Delta = 0$  since no extra decoding overhead is needed before each memory bank. Therefore, our algorithm will try to find best factors for cyclic partitioning which introduce the least decoding logics. It is obvious that all linear coefficients of array subscripts are good candidates. After all cyclic partitioning results are obtained, they are compared to the optimal result generated by the algorithm in Poncino et al. [2000] to obtain the final solution.

## 8. EXPERIMENTAL RESULTS

### 8.1. Experiment Setup

Our proposed automatic memory partitioned algorithm (*AMP*) has been implemented in our behavioral synthesis system [Cong et al. 2006]. The entire design flow is shown in Figure 12. Our algorithm takes loop kernels in behavioral languages like C as input and parses them into control data flow graphs. The synthesis engine will then perform the memory partitioning synthesis flow to improve throughput with certain design constraints. The synthesis results are dumped into RT-level VHDLs and accepted by the downstream RTL synthesis tools.

Our test cases include a set of real-life data-intensive and computation-intensive kernels: FIR, IDCT, LITHO, MATMUL, MOTEST and PALIN. FIR, IDCT, MATMUL and MOTEST are common transformation algorithms in the multi-media domain. LITHO is the  $4 \times 4$  tiling version of the work in [Cong and Zou 2008]. PALIN is a palindrome checking program in the biology domain. All of these programs have abundant memory accesses and large data parallelism, and are perfect examples for testing our partitioning algorithm. These loop kernels contain multidimensional arrays, various array access patterns including indirect array accesses and up to 50 array accesses inside the loop body.

Table I. Test Result of the LITHO Design

	II	LAT	RAMB	LUT	FF	SLICE	CP(ns)
Original	16	1603	164	1667	584	1220	9.996
Manual	1	113	184	2839	1822	2027	9.939
AMP(1)	1	112	176	2749	1941	2066	9.843
AMP(2)	2	211	168	2018	714	1583	9.984
AMP(4)	4	407	164	1890	727	1435	9.811

Table II. Performance Optimization Results on All Test Benches

	II	II(p)	Slices	Slices(p)	Overhead
FIR	3	1	241	510	2.12×
IDCT	4	1	354	359	1.01×
LITHO	16	1	1220	2066	1.69×
MATMUL	4	1	211	406	1.92×
MOTEST	5	1	832	961	1.16×
PALIN	2	1	84	65	0.77×
AVG		5.6×			1.45×

## 8.2. Case Study: Lithography Simulation

The lithography simulation program is tested to illustrate the effectiveness of our approach. Our experiments use the Xilinx Virtex-4 FPGA and ISE 9.1 tool,<sup>3</sup> and the memory port limitation is 2 for BlockRAM on the Vitex-4 platform. The original program is optimized by loop tiling, with a  $4 \times 4$  tile size. Therefore, the image array is accessed 32 times in the loop body, and the original program can only achieve  $II = 16$  as the lower bound with a 2-port BlockRAM. For comparison, we also developed a program manually with the idea in Cong and Zou [2008]. The test results are shown in Table I.

Our *AMP* algorithm automatically generates a set of feasible solutions based on different *II* constraints, with  $4\times$  difference in throughput and 60% difference in area (rows 3 – 5 in Table I show three different implementations with *II*s in parentheses). The manual design can also achieve  $II = 1$ , however, it requires substantial time to rewrite the old program, and the modified C code has almost 1000 lines in our implementation compared to about 30 lines of code in the original program. The Block RAM number increases because the memory size cannot be perfectly divided by Block RAM size, and there are more wasted internal segments with more memory banks.

Also, the results show that the QoR of our automatic approach is comparable with the manual optimized design, although the partitioning solution is not exactly the same. Our approach can generate a set of solutions automatically without changing a single line of original C code, this is especially useful for fast design space exploration like trying different tiling sizes and memory implementations. But the actual use of memory bits is not changed, since our approach does not allow overlapped partitioning solutions; this further proves the advantage of the memory partitioning algorithm over other approaches, such as increasing memory port or memory duplication.

## 8.3. Performance Optimization

Test results on all six test cases are listed in Table II. From left to right, values in each row are *II* without memory partitioning, *II* after automatic memory partitioning, number of SLICES without memory partitioning, number of SLICES after memory partitioning and the comparison between SLICE numbers.

Overall, our *AMP* algorithm can improve the throughput about  $6\times$  (up to  $16\times$ ) on average with moderate area overhead (45% area increase on average). Clock periods

<sup>3</sup><http://www.xilinx.com>.

Table III. Power Optimization Results on All Test Benches

	Est	Est(p)	CMP	Pwr	Pwr(p)	CMP
FIR	6630.7	4764.1	28%	782	572.3	26%
IDCT	280	136	51%	5200	2900	44%
LITHO	5623.2	2520	55%	1900	1300	31%
MATMUL	62611	20160	67%	897.6	200.7	77%
MOTEST	29755	19472	34%	760	680	10%
PALIN	3369	3369	0%	66.3	66.3	0%
AVG			39.5%			31.8%

have marginal changes after memory partitioning and are omitted in the final results for that reason. With higher throughput, resource sharing opportunities are less than the original design, which is the main reason for the area overhead. However, latency of the loop body is reduced in most cases with more memory bandwidth; therefore we can even find area reduction on one test case (PALIN) after memory partitioning. Overall, the latency reduction of the loop body compromises the overhead of tighter resource sharing, which suggests that the overall area overhead is acceptable compared to the dramatic throughput improvement.

Our algorithm can find solutions for all these designs within 1 minute on a 2.4GHz Pentium 4 Linux PC since the number of array accesses in each design is not very large. An extreme case of the lithography simulation program is tested to further study the scalability of our algorithm. The new case has a  $16 \times 16$  tile size, resulting in about 500 array accesses on the image array. Our algorithm found the optimal memory partitioning solution ( $16 \times 16$  cyclic partitioning) in 175 seconds, and the pipelining algorithm took another 428 seconds to finish. In theory, our algorithm can improve the throughput by  $256\times$  in this case; however, it cannot be mapped into even the largest FPGA Virtex-4 board because of the huge number of Block RAMs required by this implementation. In general, practical designs will not have a very large amount of array accesses and arithmetic operations due to cost requirements, and they should be handled perfectly by our algorithm.

#### 8.4. Power Optimization

The effect of power optimization is further evaluated in this section. As we mentioned, the power optimization step tries to reduce the total power consumption on the throughput optimization results. The same set of benchmarks is used here. However the Magma Talus RTL to GDSII synthesis tool<sup>4</sup> is used in this experiment since the Xilinx BlockRAM is inherently partitioned with a fixed size and is not suitable for a fair comparison. A memory generator for the TSMC 90nm library<sup>5</sup> is used to generate RAM blocks. The power-size curve for one memory read access is shown in Figure 10.

Test results after memory partitioning are shown in Table III. For each line, values from left to right are estimated energy consumption of design after throughput optimization, estimated energy consumption after power optimization, comparison, and the power data reported by the Magma tool for the design after throughput optimization, the design after power optimization and comparison. The unit of data reported by Magma is  $\mu\text{W}$ . Overall, our approach can effectively reduce the dynamic power consumption about 40% based on the estimation, and power reports by the Magma tool validate the efficacy of our approach, showing about 30% dynamic power reduction on average.

<sup>4</sup><http://www.magam-da.com>

<sup>5</sup><http://www.tsmc.com>

### 8.5. Additional Exploration

Our current approach is a partial step in the ongoing research for the optimal memory partitioning problem. While continuing searching for better methods and algorithms, we found that our current algorithm can handle most practical problems surprisingly well. As shown in Table II, all designs achieve optimal throughput. We manually inspected two famous test bench suites in SPEC2006<sup>6</sup> and MediaBench [Lee et al. 1997]. Based on our observation, most of the compute-intensive kernels have very regular array access patterns which can be handled by cyclic partitioning similar to our benchmarks. However, these programs are not synthesizable into hardware in general because of pointers and recursions. Due to the complexity of rewriting these programs, no experiments have been performed on these test benches; however we expect similar results.

### 9. SUMMARY

In this article we present an automatic memory partitioning technique which can efficiently improve throughput and reduce power consumption of pipelined loop kernels. Our approach can handle general array references in most real-world applications, and dramatically improve the productivity of hardware designs. Further, this approach is integrated in our behavioral synthesis flow to provide a complete synthesis flow that considers the specific requirements for hardware designs. To our knowledge, this is the first work which considers automatic memory partitioning at the cycle-accurate level. Our future work includes supporting more partitioning schemes and finding better search algorithms to further improve the QoR of designs.

### REFERENCES

- AGARWAL, A., KRANZ, D., AND NATARAJAN, V. 1995. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Para. Distrib. Syst.* 6, 943–962.
- ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. 1995. Software pipelining. *ACM Comput. Surv.* 27, 3, 367–432.
- ANDERSON, J. M. AND LAM, M. S. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. 112–125.
- BARADARAN, N. AND DINIZ, P. C. 2008. A compiler approach to managing storage and memory bandwidth in configurable architectures. *ACM Trans. Des. Autom. Electron. Syst.* 13, 4, 1–26.
- BOMZE, I. M., BUDINICH, M., PARDALOS, P. M., AND PELILLO, M. 1999. The maximum clique problem. In *Handbook of Combinatorial Optimization*, Kluwer Academic Publishers, 1–74.
- CONG, J., FAN, Y., HAN, G., JIANG, W., AND ZHANG, Z. 2006. Platform-based behavior-level and system-level synthesis. In *Proceedings of the IEEE Systems on Chip Conference (SOCC)*.
- CONG, J., JIANG, W., LIU, B., AND ZOU, Y. 2009. Automatic memory partitioning and scheduling for throughput and power optimization. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.
- CONG, J. AND ZOU, Y. 2008. Lithographic aerial image simulation with FPGA-based hardware acceleration. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*.
- GONG, W., WANG, G., AND KASTNER, R. 2005. Storage assignment during high-level synthesis for configurable architectures. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.
- GUPTA, S. AND NAJM, F. N. 1997. Power macromodeling for high level power estimation. In *Proceedings of the 34th Design Automation Conference (DAC)*. ACM, New York, NY, 365–370.
- KENNEDY, K. AND ALLEN, J. R. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.
- LANDMAN, P. E. AND RABAAY, J. M. 1995. Architectural power analysis: the dual bit type method. *IEEE Trans. VLSI Syst.* 3, 2, 173–187.

<sup>6</sup><http://pec.it.miami.edu/cpu2006/>

- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the International Symposium on Microarchitecture*. 330–335.
- LYUH, C.-G. AND KIM, T. 2004. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *Proceedings of the Design Automation Conference (DAC)*.
- PONCINO, M., BENINI, L., AND MACII, A. 2000. A recursive algorithm for low-power memory partitioning. In *Proceedings of IEEE/ACM International Symposium on Low Power Electronics and Design*. 78–83.
- RAMANUJAM, J. AND SADAYAPPAN, P. 1991. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parall. Distrib. Syst.* 2, 4, 472–482.
- RAU, B. R. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*.
- TATSUMI, Y. AND MATTAUSCH, H. 1999. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Lett.* 35, 25, 2185–2187.
- VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. 2007. Counting integer points in parametric polytopes using barvinok’s rational functions. *Algorithmica* 48, 1, 37–66.
- WANG, Z. AND HU, X. S. 2004. Power aware variable partitioning and instruction scheduling for multiple memory banks. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*.

Received December 2009; revised June 2010; accepted November 2010