

Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization

Jason Cong*, Wei Jiang, Bin Liu and Yi Zou

Computer Science Department

University of California, Los Angeles, CA 90095, USA

Email: {cong, wjiang, bliu, zouyi}@cs.ucla.edu

ABSTRACT

Hardware acceleration is crucial in modern embedded system design to meet the explosive demands on performance and cost. 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 to reach 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 requirement. 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. Experimental results show an average 6X 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 with previous approaches, our technique can statically compute memory access frequencies in polynomial time with little to none profiling. Experimental results show about 30% power reduction on the same set of benchmarks.

Categories and Subject Descriptors

B.5.2 [Hardware]: Design Aids—*automatic synthesis*

General Terms

Algorithm, Design, Experimentation

Keywords

Behavioral Synthesis, Memory Partitioning

*Dr. Cong also serves as the Chief Technology Advisor for AutoESL Design Technologies, Inc.

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICCAD'09, November 2-5, 2009, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-800-1/09/11 ...\$10.00.

1. INTRODUCTION

Behavioral synthesis has emerged as a promising direction in EDA because of the exponentially increasing complexity in modern SoC designs. One important application of behavioral synthesis is hardware acceleration, which plays a crucial role in meeting the demanding performance requirements. Typical applications for hardware acceleration are data-intensive or computation-intensive kernels in multimedia processing, which require high throughput. Usually computation kernels are captured by perfect loop nests. However, memory bandwidth limits potential performance gain even with large parallelism available.

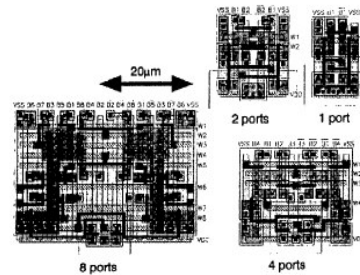


Figure 1: Area of SRAM cells (taken from [16]).

Memory bottleneck problems have drawn intensive interest, both in software and hardware designs. A natural way to tackle this problem is to increase memory ports in the final memory implementations. However, recent research in [16] 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 [10] 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 as it is hard 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. Authors in [14, 2] developed algorithms to generate communication-free data partitions for multi-computers with local memory. The approach in [1] presented an optimal tiling and iteration/data space mapping algorithm even if communication-free partitioning solutions do not exist. In general, these algorithms at-

tempt to exploit coarse-level parallelism to generate good data partitions which minimize remote data accesses. However in hardware design, data accesses must be considered at cycle-accurate level to avoid excessive simultaneously accesses on the same memory. In behavioral synthesis, the work in [7] studied the memory partitioning problem based on the footprint of data access; they can only support either row-based partitioning or column-based partitioning. Another approach in [3] 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 regiments on pipelined perfect loop nests.

Memory partitioning can also be applied for power optimization. As IC technology scales into 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 the overhead brought by performance optimization. Memory partitioning techniques for power reduction have been studied by various works [12, 18, 13]. Approaches in [12, 18] 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 [13] 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, no profiling (for affine array accesses) or lightweight profiling is needed with a parameterized polytope model of array accesses. In addition, our approach can support more partitioning schemes.

In this paper, a complete behavioral synthesis flow is proposed to solve the memory partitioning problem for throughput and power optimization. Our contributions include: (i) A complete 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. (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 which considers more general partition schemes and requires less profiling effort.

The remainder of the paper is organized as follows. Section 2 gives a motivation example for our memory partitioning problem. Section 3 describes the preliminaries and problem formulation of our memory partitioning problem, and Section 4 presents our proposed algorithms to solve this problem. Section 5 reports experimental results and is followed by conclusions in Section 6.

2. MOTIVATION EXAMPLE

The example being discussed in this section is taken from a recent work on FPGA acceleration [6]. 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 pseudo-code with a 2X2 tile size is shown in Figure 2.

In a synchronous loop pipelining algorithm, throughput is usu-

```

.....
//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];
  }
.....

```

Figure 2: Pseudo-code of lithology simulation.

ally measured by initiation interval (II) which defines the number of clock cycles that must elapse before operations of the next loop iteration can be issued. And $II \geq \text{operations/resources}$ since all operations at least execute once in II clock cycles. For the lithology simulation problem, if we simply map each array into one on-chip memory on FPGA, port limitations determine 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 2X2 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 2 on array K will always be distributed to different banks as shown in Figure 3, and as a result, the final II can be reduced to 1 after partitioning.

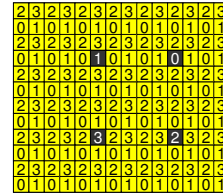


Figure 3: 2X2 partitioning and 4 array accesses on array K .

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

```

EXAMPLE 1.      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 4(a) (the dotted rectangle is the loop body, the element with lighter color is accessed by $K[i]$, and darker color by $K[i+5]$). 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 4(b). So 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 paper is that our memory partitioning approach is combined with memory access scheduling in a cycle-accurate way compared to previous approaches.

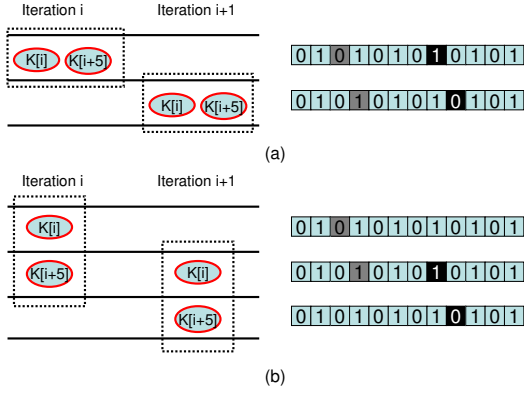


Figure 4: Impact of scheduling on memory partitioning.

3. PROBLEM FORMULATION

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. 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 $I = (i_1, i_2, \dots, i_l)$. 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 1. 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 $R_k(I) = M \cdot I + V$, where M is an $n \times l$ matrix and V is a constant vector of size n .

EXAMPLE 2. array access $R_1 : A[i-1, i+j]$ is an affine array access with respect to iteration space $\mathbb{I} = (i, j)$ since

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

In the following discussion, all array accesses are assumed to be affine array accesses. However, our approach can handle certain non-affine array accesses by extending our algorithm, as shown in Section 4.3.

DEFINITION 2. Memory partitioning is described as a function P which maps array address $A = (a_1, a_2, \dots, a_n)$ in array spaces to partitioned memory banks, i.e., $P(A)$ is the memory bank index that A belongs to after partitioning.

EXAMPLE 3. N -way 1-dimensional cyclic partitioning:

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

Loop pipelining is the most commonly used technique to improve throughput, which allows simultaneous execution of operations in several continuous loop iterations. For memory access, if the memory port number is N , and there are K memory accesses, the final II is limited to $II \geq K/N$. To overcome the memory bandwidth problem, an automatic memory partition algorithm is developed in this paper.

In a synchronous pipelined loop, 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 ($t_k \equiv t_l \bmod II$). 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 $i - \lfloor t_k/II \rfloor$. For this reason, we call $-\lfloor t_k/II \rfloor$ as the offset of loop iteration for array access R_k , and denote it as $\Delta_k(t_k, II)$.

EXAMPLE 4. For array access $K[i+5]$ (with $t_{K[i+5]} = 1$) in Figure 4(b), $\Delta_{K[i+5]} = -t_{K[i+5]}/II = -1/1 = -1$, which means if array access $K[i]$ is executed at iteration i , $K[i+5]$ in iteration $i-1$ is executed at the same clock cycle.

Based on this observation, a throughput optimization problem which considers cycle-accurate memory access patterns is formulated in Problem 1.

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 as well as scheduling steps t_k for each R_k , such that:

$$\forall I \in \mathbb{I} \quad \max(|S_{j,t}(I)|) \leq N$$

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

As mentioned earlier, 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.

EXAMPLE 5. Let $R_1 = K[i]$ and $R_2 = K[i+5]$ for array accesses in Figure 4(a), and target $II = 1$, it is easy to see that $\Delta_1 = \Delta_2 = 0$. 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 4(b), which means $II = 2$ under the same cyclic partition. Therefore, 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 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 energy consumption function $C(P)$, the goal is to find a partition P such that throughput constraints are met and overall energy is minimized.

The above problem 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 partitioning result from each loop nest.

4. PARTITIONING ALGORITHM

In this section, a novel algorithm is presented to effectively solve the memory partitioning problem. 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 5. 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. 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. After throughput constraints are met, partitioning results will be passed to the power optimization step to minimize power consumption.

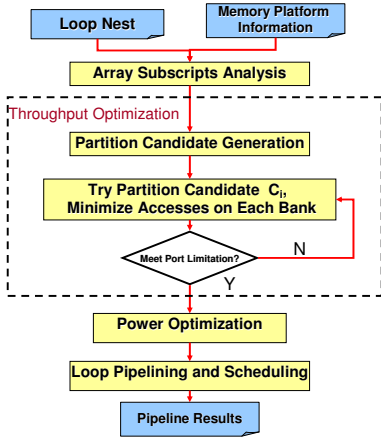


Figure 5: Memory partitioning algorithm overview.

4.1 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 majority array accesses in loop kernels are simple and regular based on the research in [11].

As mentioned above, we assume that all array access are affine in this section. Our discussion starts from a simple case: two array accesses R_1 and R_2 on a one-dimensional array in a one-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 access will never access the same memory bank. If $a_1 \neq a_2$, the following theorem provides a criteria to select 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 is proved above. \square

EXAMPLE 6. 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.

4.2 Throughput Optimization

After partitioning candidates set C on every array dimension are generated, and 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. After partitioning candidates have been generated, the partitioning algorithm attempts to solve a sub-problem of Problem 1:

PROBLEM 3. 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:

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

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

The above problem is similar to Problem 1 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 3 by introducing the concept of conflict graph.

DEFINITION 3. 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:

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

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 4.1. The conflict graph is more conservative since if two array access conflict in *any* iteration, these two operations will be considered conflicting at *every* loop iteration. The conflict graph depends on scheduling results.

EXAMPLE 7. The loop in Figure 6(a) is initially scheduled as Figure 6(b), and the corresponding conflict graph is shown in Figure 6(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 $\forall i \quad i \equiv (3 * (i + 1) + 1) \bmod 2$. The other edges of this conflict graph can be inferred in a similar way.

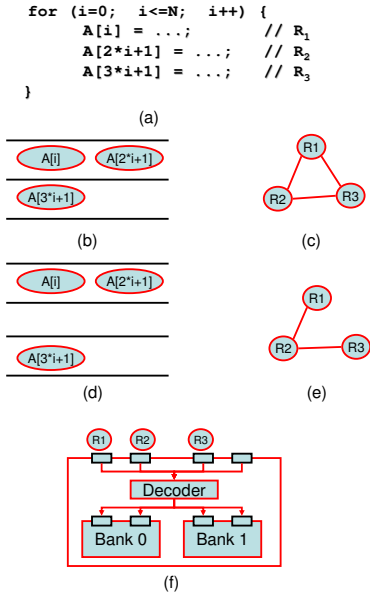


Figure 6: An illustration example.

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 3:

PROBLEM 4. Given K array references R_k on the same array A , the target throughput requirement \mathbb{I} , 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 [4]. Fortunately for our problem, the number of array accesses is under that scale. 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 of G .

The pseudo-code of our algorithm for Problem 4 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 dimension is combined with candidates on other 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, the solution is returned. Otherwise, a rescheduling step will attempt to reduce $|MC|$ as shown in line 13. The rescheduling step then reschedules nodes in MC ; each node will be tentatively moved to other 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 17.

EXAMPLE 8. Let $N = 2$ for the example in Figure 6, Figure 6(b) is the initial scheduling, and $|MC| = 3 > N$ for the conflict graph in Figure 6(c). A rescheduling result is shown in Figure 6(d), and the corresponding conflict graph is in Figure 6(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 $\mathbb{I} = 1$. The final memory implementation and port binding are shown in Figure 6(f); these 3 memory accesses can execute at the same clock cycle since at most 2 of them can access a same bank.

Algorithm 1 Search Algorithm

- 1: \mathbb{C} \rightarrow partition candidates
 - 2: $\mathbb{R} = \{R_1, R_2, \dots, R_K\}$ \rightarrow array accesses
 - 3: \mathbb{S} \rightarrow set of partition candidates on different dimensions
 - 4: \mathbb{T} \rightarrow scheduling results
 - 5:
 - 6: $\mathbb{T} = \text{get_initial_schedule_solution}()$
 - 7: **while** more candidates to search **do**
 - 8: **for all** $c_i \in \mathbb{C}$ **do**
 - 9: update current partition solution \mathbb{S}' by combining c_i
 - 10: update_conflict_graph($G, \mathbb{R}, \mathbb{T}, \mathbb{S}'$)
 - 11: $MC = \text{maximum_clique}(G)$
 - 12: **while** $|MC| > N$ and $|MC|$ decreases **do**
 - 13: $\mathbb{T} = \text{reschedule_nodes}(MC)$
 - 14: update_conflict_graph($G, \mathbb{R}, \mathbb{T}, \mathbb{S}'$)
 - 15: $MC = \text{maximum_clique}(G)$
 - 16: **end while**
 - 17: compare cost of current solution $\text{cost}(\mathbb{S}')$ with $\text{cost}(\mathbb{S})$
 - 18: update \mathbb{S}
 - 19: **end for**
 - 20: **end while**
 - 21: return \mathbb{S}
-

After memory access partitioning and scheduling step finishes, an enhanced loop pipelining algorithm is proposed under the direction of the memory partitioning algorithm to obtain the final scheduling of all operations. A classic loop pipelining algorithm called iterative modulo scheduling in [15] is adapted to generate high-quality scheduling results by exploiting the conflict-graph concept above to track resource usages after memory partitioning. Details are omitted due to page limit.

4.3 Handle Irregular Array Access

Our approach can be extended to handle certain irregular array accesses like indirect access in the lithology simulation. To handle irregular array subscripts, each irregular element in the expression tree is added to the iteration space as a pseudo-induction 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 9. For lithology simulation program in Figure 2, irregular array access address $\text{addr}_x = 10 * x - R[n] + c$ (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 $\text{addr}_x = 10 * x + z + c$ is affine with respect to I' .

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

DEFINITION 4. 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.

4.4 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 previous throughput optimization algorithm will be further partitioned to reduce power, and throughput will not be adversely impacted by the power optimization step. Our approach uses a similar approach as the work in [13]. The goal is to group frequently visited elements into small memory banks, because average power consumption decreases with memory size. The algorithm in this section solves the following problem:

PROBLEM 5. 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 $E(s)$ and a cost function of adding memory banks $CB(n)$, the goal is to find a partition P such that overall energy is minimized.

Our algorithm is developed in a similar way to the work in [13], and the optimization objective is formulated as the following:

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

$E(s)$ 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 $CB(n)$ 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 . Compared with [13], our approach considers more general partition schemes and requires less profiling effort.

Array access profile gives detail access information of each array element. In our approach, the loop iteration space is described by a polytope model which is widely used in the compiler domain.

DEFINITION 5. : 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:

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

EXAMPLE 10.

$$\begin{aligned} & \text{for } (i=0; i \leq N; i++) \\ & \text{for } (j=i; j \leq M; j++) \\ & t += A[i+j]; \end{aligned}$$

Iteration spaces and array spaces can be modeled as integer points in rational polytopes. If R_k is an affine array access $R_k(I) = M * I + V$, the number of iterations which access array address A is the number of integer point inside the intersection of iteration space \mathbb{I} and the hyperplane $\{I \mid A = M * I + V\}$. For the example above, the number of loop iterations which load $A[a]$ equals the number of integer point inside polytope:

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

The corresponding values of A , B , \vec{p} and c in Definition 5 can also be calculated from the form above:

$$PT_a = \{ I \in Q^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} * [a] + \begin{bmatrix} 0 \\ -N \\ 0 \\ -M \\ 0 \\ 0 \end{bmatrix} \}$$

DEFINITION 6. A rational n -periodic number $U(\vec{p})$ is a function $Z^n \rightarrow Q$, such that there exist a period $\vec{q} = (q_1, \dots, q_n)$:

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

A periodic number is often represented by the fracture function $\{x\} = x - \lfloor x \rfloor$. A quasi-polynomial is a polynomial with periodic numbers as coefficients.

EXAMPLE 11. $\{N/2\}$ is 1-periodic number $(0, 0.5)$ with a period 2, and $\{N/2\} \cdot N + 1$ is a quasi-polynomial.

Counting integer point inside a polytope is a fundamental problem in mathematics, and has wide application in program transformations and optimizations. The *barvinok* library[17] can solve this problem in polynomial time, and the solution is a quasi-polynomial.

Number of array accesses on $A[a_1]$ is calculated by the *barvinok* library, and 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 above equation. If N and M is unknown, users only need to provide information on these variables. Compared with [13], our approach requires only small efforts from users to get array access profile, which is a huge advantage to improve the design productivity and broaden design space exploration. Our approach is especially useful for multi-media programs or DSP designs, where most of array accesses are affine and no profiling is needed subsequently.

After array profile $f(A)$ is obtained, an iterative search step will find partitioning solutions based on the platform information. Overall, our approach uses a similar approach with [13] to search for the optimal solution of Equation 1. However, optimal solution is limited to one specific SRAM architecture called "segment configuration" in [13]. In this mode, 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, like *cyclic* partitioning. The benefit of cyclic partitioning can be illustrated by the following example:

EXAMPLE 12. $for(i=0; i \leq N; i++)$
 $t += A[2*i] + A[2*i+1];$

It is clearly that those two array accesses will be split to bank 0 and 1 respectively for partitioning $P(a) = a \bmod 2$. Therefore, no decoder is needed compared with the "segment configuration" architecture, while only one bank is activated for each original array access. Therefore, our algorithm will try to find best factors for cyclic partitioning which introduces the least decoding logics. It is obviously 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 [13] to obtain the final solution.

5. EXPERIMENTAL RESULTS

5.1 Experiment Setup

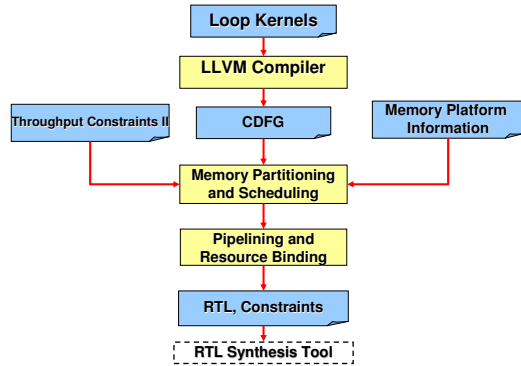


Figure 7: Implementation flow.

Our proposed automatic memory partitioned algorithm (*AMP*) has been implemented in our behavior synthesis system [5]. The entire design flow is shown in Figure 7. 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 4X4 tiling version of the work in [6]. 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.

5.2 Case study: Lithology Simulation

The lithology simulation program is tested to illustrate the effectiveness of our approach. Our experiments use the Xilinx Virtex-4 FPGA and ISE 9.1 tool [10], and the memory port limitation is 2 for BlockRAM on the Vitex-4 platform. The original program is

Table 1: 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

optimized by loop tiling, with a 4X4 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 2-port BlockRAM. For comparison, we also developed a program manually with the idea in [6]. The test results are shown in Table 1.

Our *AMP* algorithm automatically generates a set of feasible solutions based on different II constraints, with 4X difference in throughput and 60% difference in area (Rows 3 – 5 in Table 1 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 change a single line of original C code, this is especially useful for fast design space exploration like trying different tiling sizes and memory implementations.

5.3 Performance Optimization

Test results on all six test cases are listed in Table 2. 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.

Table 2: Performance optimization results on all test benches.

	II	II(p)	Slices	Slices(p)	Overhead
FIR	3	1	241	510	2.12x
IDCT	4	1	354	359	1.01x
LITHO	16	1	1220	2066	1.69x
MATMUL	4	1	211	406	1.92x
MOTEST	5	1	832	961	1.16x
PALIN	2	1	84	65	0.77x
AVG		5.67x			1.45x

Overall, our *AMP* algorithm can improve the throughput about 6X (up to 16X) on average with moderate area overhead (45% area increase on average). Clock periods 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 lithology simulation program is tested to further study the scalability of our algorithm. The new case has a 16X16 tile size, resulting in about 500 array accesses on the image array. Our algorithm found the optimal memory partitioning solution (16X16 cyclic partitioning) in 175 seconds, and the pipelining algorithm took another 428 seconds to finish. In theory, our algorithm can improve the throughput by 256X on this case, however it cannot be mapped into even the largest FPGA Virtex-4 board because of the huge number of Block RAMs. In general, practical designs would not have a very large amount of array accesses and arithmetic operations due to cost requirements, and they should be perfectly handled by our algorithm.

5.4 Power Optimization

The effect of power optimization is further evaluated in this section. The same set of benchmarks is used here. However Magma talus RTL to GDSII synthesis tool [8] is used in this experiment since Xilinx BlockRAM is inherently partitioned with fixed size and not suitable for fair comparison. A memory generator for TSMC 90nm library [9] is used to generate RAM blocks. The power-size curve for one memory read access is shown in Figure 8.

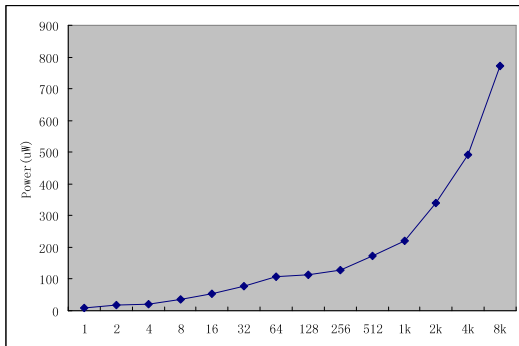


Figure 8: Power consumption of a memory read access against the memory size.

Table 3: 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%

Test results after memory partitioning is shown in Table 3. For each line, values from left to right are estimated energy consumption of original design, estimated energy consumption after partitioning, comparison, power date reported by Magma tool for the original design, after partitioning and comparison. Unit of data reported by Magma are μW . Overall, our approach can effectively reduce the dynamic power consumption about 40% based on the estimation, and power reports by Magma tool validate the efficacy of our approach, showing about 30% power reduction on average.

6. CONCLUSIONS

In this paper we present an automatic memory partitioning technique which can efficiently improve throughput and reduce power consumption of pipelined loop kernels. To our knowledge, this is the first work which consider automatic memory partitioning at the cycle-accurate level.

7. ACKNOWLEDGMENT

This work is partially supported by GSRC, the GRC Contract 2009-TJ-1879, and the NSF grant CNS-0725354. Support from Magma and AutoESL inc. is also greatly acknowledged.

8. REFERENCES

- [1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6:943–962, 1995.
- [2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *In Proc. of the SIGPLAN PLDI'93*, pages 112–125, 1993.
- [3] N. Baradaran and P. C. Diniz. A compiler approach to managing storage and memory bandwidth in configurable architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 13(4):1–26, 2008.
- [4] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.
- [5] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *Proceedings of IEEE SOCC*, 2006.
- [6] J. Cong and Y. Zou. Lithographic aerial image simulation with FPGA-based hardware acceleration. In *Proc. of international symposium on Field programmable gate arrays*, 2008.
- [7] W. Gong, G. Wang, and R. Kastner. Storage assignment during high-level synthesis for configurable architectures. In *Proc. of ICCAD '05*.
- [8] <http://www.magam-da.com>. *Magma Website*.
- [9] <http://www.tsmc.com>. *TSMC Website*.
- [10] <http://www.xilinx.com>. *Xilinx Website*.
- [11] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.
- [12] C.-G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *Proc. of DAC '04*.
- [13] M. Poncino, L. Benini, and A. Macii. A recursive algorithm for low-power memory partitioning. In *Proc. of IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 78–83, 2000.
- [14] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):472–482, 1991.
- [15] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994.
- [16] Y. Tatsumi and H. Mattausch. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Letters*, 35(25):2185–2187, Dec 1999.
- [17] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.
- [18] Z. Wang and X. S. Hu. Power aware variable partitioning and instruction scheduling for multiple memory banks. In *Proc. of DATE '04*.