OPERATION SCHEDULING FOR FPGA-BASED RECONFIGURABLE COMPUTERS

Colin Yu Lin, Ngai Wong and Hayden Kwok-Hay So

Department of Electrical and Electronic Engineering
University of Hong Kong, Hong Kong
{linyu, nwong, hso}@eee.hku.hk

ABSTRACT

Many high-performance applications involve large data sets that are impossible to fit entirely within on-chip memories of even the largest FPGAs. As a result, they must be stored in off-chip SDRAMs and loaded onto the FPGAs as computations progress. Because of the high latency and energy consumption associated with off-chip memory accesses, it is important to develop efficient operation schedules that not only minimize latency of computations, but also the amount of data I/Os. We formulate this problem as a modified resource-constrained job scheduling problem. The problem is then solved using a list scheduling algorithm that takes advantage of the fast burst-mode access of SDRAMs. Results have shown that for large problem sizes, the performance of our algorithm is within 1% of a hand-optimized matrix-matrix multiplication implementation, with zero memory overhead, and is within 0.03% of the theoretical minimum latency of an 8-by-8 cofactor matrix computation.

1. INTRODUCTION

One characteristic of high-performance applications is that most of their computations involve extremely large datasets. When FPGAs are used to accelerate such computations, data must be stored in off-chip memories and be loaded onto the FPGA as the computations progress, using on-chip memories as temporary storages.

Most existing scheduling algorithms focus on minimizing computation latency of an application. However, as power consumption has become one of the most critical issues even for high-performance computers, it is important to develop scheduling algorithms that take power consumptions into consideration. Comparing to a conventional processor-based system, an FPGA-based reconfigurable computer may attach more than one external memories to an FPGA [1]. However, accessing data in external memories through very wide buses incurs significant power overhead when compared to internal operations. As a result, on top of minimizing computation latency, it is also important to minimize the number of data I/O operations, possibly through a different schedule of the compute operations.

One way to achieve efficient operation schedule is to develop carefully hand-optimized architecture and operation schedule for the target computation. For example, a systolic array structure was used in [2] for which energy and latency of the matrix operations are analyzed. Although such hand-optimized solutions usually provide near-optimal schedule for a particular problem, it is difficult to apply such technique generically to all applications.

In this work, we address this operation scheduling problem using a time-indexed integer linear programming (ILP) formulation based on the well-studied resource-constrained project scheduling problem (RCPSP) [3]. The difference between our approach and an RCPSP is that an operation, in particular a data input operation, may be executed more than once depending on the schedule. Also, we minimize not only the overall latency, but also the number of data I/Os. Furthermore, taking advantage of the inherent asymmetric access speed of modern SDRAMs, burst mode transfers from external memories is utilized as much as possible.

Solving such ILP is, in general, computationally intractable for even moderately sized problems. As a result, we have developed a variation of the Graham's list scheduling algorithm [4] that gives approximation to the optimal solution.

In Section 2, we first describe our scheduling ILP model. Then in Section 3, the list scheduling algorithm that solves our scheduling model is presented. In Section 4, we illustrate the operation and performance of our algorithm using the tasks of large matrix multiplications and calculating cofactor matrices. Finally, we conclude the paper in Section 5.

2. SCHEDULING MODEL

Our scheduling model begins by defining a directed acyclic graph $G(V, E)$ that augments the data flow graph (DFG) of the target computation. Two types of nodes, V_{op} and V_{data}, are defined, where $V = V_{op} \cup V_{data}$. A node is defined in V_{op} for each elementary operation such as add, multiply or multiply-accumulate. A node is defined in V_{data} for each data element in the input matrices, acting as a data source. An edge (i, j) is drawn from node i to node j if operation j depends on data from operation or data source i.

Each data source node in V_{data} is labeled d_i where $i \in \{1 \ldots |V_{data}|\}$. Each operation node in V_{op} is labeled o_j where $j \in \{1 \ldots |V_{op}|\}$. F contains the set of all operations that have no successor in the DFG.

2.1. Architecture Assumptions

In order to limit our scope, we make certain assumptions about the target reconfigurable system. In our model, only one FPGA and two off-chip memory devices are considered. One off-chip memory is used for input data while the other one used for output. At any one point in time, the maximum number of words that the FPGA can read from memory and/or write to memory is the bandwidth $\mathbf{B}$. All data in the input matrices have the size of a word.

We assume $\mathbf{P}$ processing elements (PEs) are implemented. In practice, these PEs may be implemented using either on-chip reconfigurable logic or dedicated DSP blocks. The PEs are programmable and may be reconfigured to perform either add or multiply in each step.

There are $\mathbf{M}$ words of on-chip memory that act as local buffers. A PE may access any memory location and all PEs may access the memory at the same time. In practice, they may be implemented as distributed memory or a distribution network that broadcasts values to PEs.

Furthermore, we assume that I/O operations and the PEs may operate in parallel.

2.2. Optimization Goal

The goal of our ILP model is to minimize system latency while minimizing I/O operations.

System latency is determined by the computation time and the I/O time of the system. The computation time of the system is related to the number of operations required and the number of PEs available. However, in practice, most FPGA applications that require complex computations are limited by $\mathbf{B}$, $\mathbf{M}$ and $\mathbf{P}$. The goal of our model is therefore to devise the optimal schedule for carrying out all necessary operations under these constraints. Note that in our model, we assume I/O and computation may be conducted in parallel, thereby hiding a lot of the latency from reading external memory. Furthermore, a second optimization goal for our model is to minimize data I/Os.

For ease of explanation, each elementary operation to be scheduled, as well as the time to read a data from off-chip memory is assumed to be identical. Burst mode transfer from memory will be considered in Section 3 when the list scheduling algorithm is devised.

2.3. ILP Model

We define the following variables for our formulation:

1. x_i^s and $\bar{x}_i^s$ are 0-1 integer variables associated with input data d_i. $x_i^s = 1$ if input data d_i is read-in on control step s, $s \in \{1 \ldots l\}$ (l is the system latency); otherwise, $x_i^s = 0$. $\bar{x}_i^s = 1$ if input data d_i is overwritten in on-chip memory on control step s; otherwise, $\bar{x}_i^s = 0$.
2. y_j^s and $\bar{y}_j^s$ are 0-1 integer variables associated with operation o_j. $y_j^s = 1$ if operation o_j is scheduled into control step s; otherwise, $y_j^s = 0$. $y_j^s = 1$ also denotes that the result of operation o_j originates in control step s. $\bar{y}_j^s$ is used to denote that operation o_j has its result last used in control step s.

Using the above variables, the operation scheduling problem can be formulated as follows:

minimize

$$w_l l + w_p \sum_s \sum_i x_i^s \tag{1}$$

subject to

$$\forall i: \quad \sum_s x_i^s = \sum_s \bar{x}_i^s \geq 1; \tag{2}$$

$$\forall s: \quad \sum_i x_i^s \leq \mathbf{B}; \tag{3}$$

$$\forall j: \quad \sum_s y_j^s = \sum_s \bar{y}_j^s = 1; \tag{4}$$

$$\forall s: \quad \sum_j y_j^s \leq \mathbf{P}; \tag{5}$$

$$\forall o_j \in F: \quad \sum_s \left(y_j^s s\right) \leq l; \tag{6}$$

$$\forall j: \quad y_j^1 = 0; \tag{7}$$

$$\forall s > 1 \forall (d_i, o_j) \in E: \quad \sum_{t=1}^{s-1} \left(x_i^t - \bar{x}_i^t\right) \geq y_j^s; \tag{8}$$

$$\forall (o_j, o_k) \in E: \quad \sum_s \left(y_j^s s\right) + 1 \leq \sum_s \left(y_k^s s\right); \tag{9}$$

$$\forall (o_j, o_k) \in E: \quad \sum_s \left(y_k^s s\right) \leq \sum_s \left(\bar{y}_j^s s\right); \tag{10}$$

$$\forall o_j \in F: \quad y_j^s \leq \bar{y}_j^s; \tag{11}$$

$$\forall s: \quad \sum_{t=1}^{s} \sum_i \left(x_i^t - \bar{x}_i^t\right) + \sum_{t=1}^{s} \sum_j \left(y_j^t - \bar{y}_j^t\right) \leq \mathbf{M}. \tag{12}$$

The objective function in (1) states that the goal of the ILP model for operation scheduling is to schedule data input and computation with minimum system latency l and the total data input operations $\sum_s \sum_i x_i^s$. Weights w_l and w_p are used to set the latency/power scheduling strategy.

Constraints (2) and (3) are *the input data constraints*. Constraint (2) states that each input data is at least read-in once, and each data in on-chip memory is at least overwritten once. Constraint (3) is also called *the I/O bandwidth constraint*. On each control step, the number of data read-in to on-chip memory is limited by $\mathbf{B}$.

Constraints (4)-(6) are *the operation constraints*. Constraint (4) states that each operation is to be scheduled to operate exactly once. Constraint (5) is also called *the processing element constraint*. On each control step, the number of operations scheduled is limited by $\mathbf{P}$. $\sum_s \left(y_j^s s\right)$ denotes the control step s that operation o_j is scheduled into. Constraint (6) states that the system latency l is determined by the last scheduled operation in final operation set.

Constraints (7) and (8) are the *input-operation data dependency constraints*. All operations cannot be scheduled into control step 1, as in constraint (7). Constraint (8) states that if there is an input-operation data dependency between input data d_i and operation o_j, then operation o_j can only be scheduled into control step s if input data d_i is in on-chip memory in control step $s-1$.

Constraints (9)-(11) are *the operation-operation data dependency constraints*. If there is a data dependency between two operations o_j and o_k, and operation o_j is the immediate predecessor of operation o_k, operation o_k should be scheduled later than operation o_j, as in constraint (9), and the result of operation o_j should remain in on-chip memory until all successors of operation o_j are scheduled, as in constraint (10). If $o_j \in F$, operation o_j has no successor, $y_j^s \leq \bar{y}_j^s$.

Constraint (12) is *the on-chip memory constraint*. The summation $\sum_{t=1}^{s} \sum_i \left(x_i^t - \bar{x}_i^t\right)$ represents the amount of input data residing on-chip awaiting to be used. The summation $\sum_{t=1}^{s} \sum_j \left(y_j^t - \bar{y}_j^t\right)$ denotes the amount of temporary buffer needed to hold results from earlier operations that will be used in later operations. As a result, the sum of the two represents the amount of on-chip memory required at control step s, which is therefore bounded by $\mathbf{M}$.

3. LIST SCHEDULING

Because of its simplicity, list scheduling forms the basis for many approximation algorithms to different types of RCPSPs [5]. For a dependency graph with no directed cycles, list scheduling algorithms construct feasible schedules with a low computational effort of $O\left(\left|E_{op}\right|\right)$. Our list scheduling algorithm works in a similar way as Graham list scheduling, except that the priorities are updated after each step.

As can be seen from Algorithm 1, the main procedure, ListSchedule, is fairly straightforward. For each control step, $p \leq \mathbf{P}$ operations with highest priorities in the ready queue, and $b \leq \mathbf{B}$ data input with highest priorities in the ready queue are scheduled to take place. Once the operations are scheduled, priorities of nodes with the same successor of this node are updated. The effectiveness of our algorithm therefore lies with the way data and compute operations are selected in each step. This selection process depends on three factors: (i) the state of an operation; (ii) the priority of an operation; and (iii) the available system resource.

Each data and compute operation node can be in one of several states. An input data operation can be in one of two possible states, READY and DONE. Similarly, a compute operation can be in one of three states, READY, NOTREADY and DONE. In the beginning of the algorithm, all input data are READY and all operations are NOTREADY. An operation transits from NOTREADY to READY when all of its immediate input data and/or operation predecessors are scheduled. Once executed, the state of the data operation or compute operation will change to DONE.

Algorithm 1 A list scheduling algorithm for scheduling data I/O and compute operations on FPGAs.

```
procedure UpdateSchedulePriority($k$)
  for all successor $s$ of $k$ do
    update state of $s$
    if $state(s)$ = NOTREADY then
      $cnt \leftarrow 0$
      for all predecessor $p$ of $s$ do
        if $state(p)$ = NOTREADY or READY then
          $cnt \leftarrow cnt + 1$
      if $cnt = 1$ then
        $spri(p) \leftarrow spri(p) + 1$

procedure ComputeMemRelPriority($r$)
  for all successor $s$ of $r$ do
    $cnt \leftarrow 0$
    if $state(s)$ = READY then
      $cnt \leftarrow cnt + 1$
  $mpri(r) \leftarrow cnt$

procedure ListSchedule
  $spri(i) \leftarrow 0 \quad \forall i \in V_{data}$
  $spri(j) \leftarrow 0 \quad \forall j \in V_{op}$
  while not all operations are scheduled do
    schedule $p \leq \mathbf{P}$ operations with highest priorities
    for all scheduled operations $j$ do
      UpdateSchedulePriority($j$)
    schedule $b \leq \mathbf{B}$ input data with highest priorities
    for all scheduled data $i$ do
      UpdateSchedulePriority($i$)
    if on-chip memory is not adequate then
      for all on chip data and operation results $r$ do
        ComputeMemRelPriority($r$)
      store out $m$ data/results with lowest priorities
```

The priority of an input data or operation is updated in the procedure UpdateSchedulePriority. Initially, priorities of all data and compute operations are set to 0. When an input data or operation is scheduled, it will affect the state of its successors. Some of its immediate operation successors will become READY, while some of them remain NOTREADY. For each of these NOTREADY operations, check their immediate predecessors. If only one immediate input data or operation predecessor is not DONE, then its priority will be increased by 1. It is done so because once this input data or operation is scheduled and its state becomes DONE, one operation will immediately become READY. The number of the successor operations that will become READY is impor-

Table 1. Matrix-matrix multiplications using list scheduling algorithm (**L**) and systolic array implementation (**S**) [2]. The overheads are listed under the column **O(%)**. $\mathbf{P} = 40, \mathbf{B} = 2, \mathbf{M} = 50000$.

orders	**latency(memory required)**		
	L	**S**	**O(%)**
40	1904 (1681)	1640 (1680)	16.10 (0.06)
80	13404 (6561)	12880 (6560)	4.07 (0.02)
120	44104 (14640)	43320 (14640)	1.81 (0.00)
160	103604 (25920)	102560 (25920)	1.02 (0.00)
180	147154 (32760)	145980 (32760)	0.80 (0.00)
200	201504 (40400)	200200 (40400)	0.65 (0.00)

tant because it increases the number of candidates that will be ready in the next step.

Finally, the selection process depends on the amount of available system resources. Obviously, the number of data I/O scheduled in one step, b, must not exceed $\mathbf{B}$. Also, the number of compute operation scheduled, p, must not exceed the minimum of $\mathbf{P}$ and the number of READY operations.

Furthermore, FPGA on-chip memory may not be enough to store all input data and temporary operation results, especially for large input datasets. As a result, a process of storing out some input data and/or partial results must be executed at the end of each scheduling step. Another priority for each on-chip data and operation result is required to determine which data/result to be stored out. Once a data/result is stored out, its successors with state READY will become NOTREADY immediately. The number of these successors is used to represent the penalty caused. As a result, the data and results with lowest penalties will be chosen.

Using list scheduling, the support for burst mode access from external memory is provided by reading consecutive data in each read from external memory.

4. RESULTS

In theory, multiplying two n-by-n matrices requires n^3 multiply accumulate operations. In [2], the hand-optimized systolic array architecture achieved a near-optimal latency of $n^3 + n$ when the available on-chip memory capacity $M \geq n^2 + 2n$. Multiplications are scheduled automatically using our list scheduling and the results are shown in Table 1.

For large n, our list scheduling algorithm can successfully schedule operations with latency and memory requirements close to the hand-optimized systolic array implementation. Although it doesn't work well for small n, the result is encouraging as no manual intervention was needed to obtain such near-optimal result.

Cofactor matrix computation is a matrix operation that is required in computations such as to find the inverse of a matrix. Although many advance algorithms have been developed because of its complexity, a naïve solution used

Table 2. Cofactor matrix computation using list scheduling algorithm (**L**) and compared to theoretical values (**T**). The overheads are in the column **O(%)**. $\mathbf{P} = n^2, \mathbf{B} = 2, \mathbf{M} = 200$.

orders	latency			memory
	L	**T**	**O(%)**	**L**
4	23	17	35.29	43
5	105	95	10.53	75
6	607	599	1.34	108
7	4332	4320	0.28	147
8	35291	35279	0.03	188

here for illustration purposes. As shown in Table 2, our list scheduling algorithm again provides near-optimal solutions.

5. CONCLUSION

In this paper, we have presented time-indexed ILP models that schedule I/O and computational operations for applications that involve large input datasets. The model generates optimal schedule given the fully expanded data dependency graph of the target operations. To solve this computationally intractable ILP problem, a list scheduling relaxation algorithm is presented. Using the relaxed list scheduling algorithm, matrix-matrix multiplications with large matrices and cofactor matrix computations are presented. Results have shown that given large enough problem sizes, our list scheduling algorithm achieves near-optimal solutions. In the case of matrix-matrix multiplication, latencies of our schedules are within 1% of the previously published hand-optimized systolic array implementation. For cofactor matrix computations, our result is within 0.03% of the theoretical minimum latency.

6. REFERENCES

[1] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Design & Test*, vol. 22, no. 2, pp. 114–125, 2005.

[2] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans. VLSI Syst.*, vol. 13, no. 11, pp. 1305–1319, 2005.

[3] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods," *European Journal of Operational Research*, vol. 112, no. 1, pp. 3 – 41, 1999.

[4] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, Mar. 1969.

[5] R. Kolisch and S. Hartman, "Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis," in *Project Scheduling: Recent Models, Algorithms, and Applications*, J. Weglarz, Ed. Springer, 1999, pp. 147–178.