Loop Pipelining and Optimization for Run Time Reconfiguration*

Kiran Bondalapati and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2562, USA.
{kiran,prasanna}@ceng.usc.edu
http://maarcII.usc.edu

Abstract. *Lack of automatic mapping techniques is a significant hurdle in obtaining high performance for general purpose computing on reconfigurable hardware. In this paper, we develop techniques for mapping loop computations from applications onto high performance pipelined configurations. Loop statements with generalized directed acyclic graph dependencies are mapped onto multiple pipeline segments. Each pipeline segment is executed for a fixed number of iterations before the hardware is reconfigured at runtime to execute the next segment. The reconfiguration cost is amortized over the multiple iterations of the execution of the loop statements. This alleviates the bottleneck of high reconfiguration overheads in current architectures. The paper describes heuristic techniques to construct pipeline configurations which have reduced total execution time including the runtime reconfiguration overheads. The performance benefits which can be achieved using our approach are illustrated by mapping example application loop onto Virtex series FPGA from Xilinx.*

1 Introduction

Reconfigurable computing has demonstrated significant performance gains for several classes of applications[5]. Application mapping onto configurable hardware still necessitates expertise in low-level hardware details. Automatic mapping of applications onto configurable hardware is necessary to deliver high performance for general purpose computing. In this paper we address the issues in mapping application loops onto reconfigurable hardware to optimize the total execution time. Total execution time includes the time spent in actual execution on the hardware and the time spent in reconfiguring the hardware.

Configurable hardware can be utilized to execute designs which are larger than the available physical resources. Run Time Reconfiguration(RTR) between computations facilitates dynamic adaptation of the hardware to suit the design area and computational requirements. But, in current devices, reconfiguration time is still significant compared to the execution time. We focus on developing

mapping techniques which exploit RTR but attempt to reduce the reconfiguration overhead. This is accomplished by amortizing the the reconfiguration overheads over the execution of large number of iterations of the loop.

Loop statements contribute to a significantly large component of the execution time of an application. Pipelined designs are well structured and map well onto configurable devices. Most reconfigurable architectures, including FPGA devices, provide excellent support for pipelining with their regular logic block layout and large number of registers [17]. Pipelined designs have reduced and predictable delays because they use mostly local interconnections. Hence, mapping loop computations onto pipelined configurations proves to be very effective on configurable hardware.

In this paper, we develop techniques to map computations in a loop onto reconfigurable hardware. The data dependencies in the loop statements constitute a directed acyclic graph (DAG). These loop statements are mapped onto pipelined configurations executing in the reconfigurable hardware. Our mapping techniques attempt to minimize the total execution cost for the computations including the reconfiguration cost. The statements are split into multiple pipeline segments which are executed sequentially for a fixed number of iterations each. Reconfiguration is performed after execution of a pipeline segment to execute the next segment.

Generating optimal schedule from a given task graph is an NP-complete problem. In this paper, heuristic algorithms are utilized to reduce the reconfiguration cost between different pipeline segments. We compare the effectiveness of our heuristics against a greedy heuristic based list scheduling. Our mapping techniques promise potential performance improvement on several classes of FPGAs. We evaluate the performance of our mapping techniques on the Virtex series FPGA from Xilinx [17].

In Section 2, we describe some related research work which addresses similar issues. Our heuristic based algorithms are described in detail in Section 3 and illustrated by using an example. In Section 4, we evaluate the performance benefits achieved using our approach. We draw conclusions based on our approach in Section 5.

2 Related Work

Pipelining of designs has been studied by several researchers in the configurable computing domain. Cadambi et. al. address the issues in mapping virtual pipelines onto a physical pipeline by using incremental reconfiguration in the context of PipeRench [6]. Luk et. al. describe pipeline morphing and virtual pipelines as an idea to reduce the reconfiguration costs [11]. A pipeline configuration is morphed into another configuration by incrementally reconfiguring stage by stage while computations are being performed in the remaining stages. Weinhardt describes the generation of pipelined circuits from parallel-FOR loops in high level programming language [15]. Weinhardt et. al. also developed pipeline vectorization techniques [16].

Other research has addressed related issues in mapping circuits onto reconfigurable hardware [2, 7, 10, 12, 14]. Our prior research has also developed other techniques for mapping application loops [1, 3, 4]. In this paper, the focus is on Run Time Reconfiguration at a different granularity. Our approach is to exploit Run Time Reconfiguration to achieve high performance but schedule it infrequently to minimize the overheads. Algorithmic pipeline construction and partial reconfiguration at runtime are exploited to achieve this goal.

3   Pipeline Construction

The speed-up that can be obtained by using configurable logic increases as the computations in a loop increase. But, the configurable resources that are available can be lower than the required resources to pipeline all the computations in the loop. In this case, the pipeline has to be segmented to run some of the pipeline stages and reconfigured to execute the remaining computations.

In this paper, we consider loops which do not have loop carried dependencies. Such loops do not have any dependencies between different iterations of the loop. Loop transformations can be applied to remove some existing loop carried dependencies. We also assume that the number of iterations to be executed is significantly larger than the number of pipeline stages. Hence, the cycles involved in filling and emptying the pipeline are insignificant compared to the actual execution cycles of the pipeline stages.

The execution of the complete loop can be decomposed into multiple segments, where a fixed number of iterations of each segment are executed in sequence starting from the first segment. Each segment consists of multiple pipeline stages. The logic is reconfigured after each segment to execute the next segment. The intermediate results from each segment execution are stored in memory. The execution of the sequence of segments is repeated until the required number of iterations of the loop are completed. We assume that the reconfiguration of the different segments is controlled by an external controller(e.g. a host processor).

3.1   Definitions

Reconfigurable Architecture A configurable logic array of size $L \times W$ and intermediate memory of size M. One of the basic goals of our approach is to exploit the on chip memory or fast access local SRAM provided in several reconfigurable architectures. M represents the size of this memory.

Input Task Specification A dependency graph $G(V, E)$ of the application tasks of the loop to be executed for N iterations. Each task node v_i denotes the operation to be performed on the inputs specified by the incoming edges to the node. The directed edge e_{ij} from v_i to v_j denotes the data dependency between the two nodes. The weight w_{ij} on each edge denotes the number of bits of data communicated between the nodes.

Output Pipeline Configuration A sequence σ of pipeline segments $\sigma_1, \sigma_2, \ldots, \sigma_p$ where each segment $\sigma_i (1 \leq i \leq p)$ consists of q number of stages $s_{i1}, s_{i2}, \ldots, s_{iq}$.

The pipeline stages are the mapping of the computational task nodes V to configurations of the device. Each of the stages s_{ij} is the configuration which executes a specific task in the input task graph. The size of a pipeline stage is given by the length l_{ij} and the width and w_{ij}. Some of the stages in each segment might be *null* stages which are not actual tasks but are just *place-holders* as explained later in Section 3.6.

Segment Clock Speed Each pipeline segment σ_i can be executed at a different clock speed f_{σ_i} depending on the maximum clock speed at which the stages in that segment can operate.

Segment Data Output A pipeline stage s_{ij} has global outputs if any of the outgoing edges from a task node are to a node that is not mapped to the same pipeline segment. The size of the segment data output $DO_i (1 \leq i \leq p)$ of all the pipeline stages in a segment $\sigma_i (1 \leq i \leq p)$ is given by the sum of all the global outputs of the stages in the segment.

Segment Iteration Count The number of iterations N_σ for which each pipeline segment is executed before reconfiguring to execute the next segment. N_σ depends on the size of the available memory to store the intermediate results. We assume that the initial and final results are communicated from/to external memory.

$$N_\sigma = min_i \left\{ \frac{M}{DO_i + DO_{i+1}} \right\} \quad 1 \leq i \leq p - 1$$

Reconfiguration Cost The reconfiguration cost R_{loop} is the total cost involved in reconfiguring between all the segments of the pipeline configuration. This includes the cost of configuring between the last segment and the first segment if $N > N_\sigma$. The reconfiguration cost between any two segments is given by the difference in the two pipeline configurations. Partial reconfiguration of the device in columns is assumed in our computation. We use the number of logic columns in which the configurations are different as the measure of the reconfiguration cost. When the corresponding stages in different segments are dissimilar, the reconfiguration cost accounts for the multiple adjacent stages that need to be reconfigured.

Total Execution Time The total execution time E is given by the sum of the execution times for each segment and the total reconfiguration time.

$$E = N * \left(\sum_{i=1}^{p} \frac{1}{f_{\sigma_i}} \right) + \frac{N}{N_\sigma} * R_{loop}$$

3.2 Phase 0: Pre-processing and Mapping

In this phase the computation tasks in the input DAG are mapped onto components of the given logic device. The components are chosen from the set of library components available for executing the given application tasks in the task graph. Different components can have different logic-area/execution-time tradeoffs and could potentially have different degrees of pipelining and footprint on the device after layout. The library component of the highest degree of pipelining which

satisfies other constraints specified by the task graph(such as precision of inputs) is chosen for a task.

Our proposed approach is illustrated using the mapping and scheduling of the N-body simulation application and the FFT butterfly computation. The resulting task graphs after Phase 0 with the dependency edges are shown in Figure 1. In the graph the operations are represented as A - Addition, M - Multiplication, S - Subtraction and Sh - Shift right by 4 bits(Divide by 16). The operations in the graph are all 16 bits so the weights on the edges are not indicated.

3.3   Phase 1: Partitioning

The partitioning phase generates multiple partitions where size of each partition is smaller than the size of the device. This phase attempts to optimize two criterion - (1) maximize the size of the partition (2) minimize the weight of the edges between partitions. The first criterion improves the logic utilization and the second criterion reduces the memory required to buffer intermediate results generated by each partition (pipeline segment). A sketch of the partitioning algorithm is given below without the intricate details.

A heuristic based multi-way partitioning is used to incrementally generate each of the partitions. The largest size node is chosen from among the list of *Ready* nodes (whose inputs have been computed) to be added to the current partition. When no more nodes can be added to the current partition, a new partition is initiated. For adding a *Ready* node v_i to a partition π_j, the heuristic uses the following sums of weights of edges:

- ω_1: weight of *in* edges to v_i from nodes in π_j
- ω_2: weight of *in* edges to v_i from nodes not in π_j
- ω_3: weight of *out* edges from v_i to nodes in π_j
- ω_4: weight of *out* edges from v_i to neighbors of π_j
    v_k is a neighbor of π_j if there is an edge from a node in π_j to v_k and $v_k \notin \pi_j$
- ω_5: weight of *out* edges from v_i to nodes not in π_j and not neighbors of π_j

The node chosen is the node with maximum value of $\omega_1 + \omega_3 + \omega_4 - \omega_2 - \omega_5$. The primary inputs and outputs are not considered in computing the weights. The largest node which fits in the current partition satisfying the above condition is added to the current partition. Ties are broken by using the height of the node and the different weights of edges listed above. The resulting partitions are illustrated by the partition number on the nodes of the graph in Figure 1.

3.4   Routing Considerations

The algorithm for the partitioning of the task graph assumes that there are enough routing resources to communicate between the different pipeline stages and from pipeline stages to the memory. Some of the pipeline stages might have
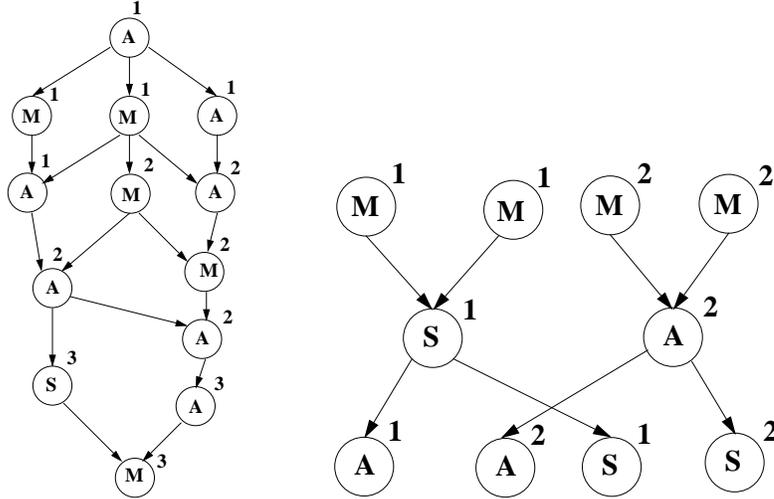
Fig. 1. (a) N-body simulation task DAG and (b) FFT task DAG with partition numbers

global inputs and outputs. These are data inputs and outputs which are not to adjacent pipeline stages, but from/to either non-adjacent stages or from/to memory. Some of the data outputs from the pipeline stages might have to be buffered (using registers) before they are consumed in the later stages.

Routing resources are an important consideration when mapping communication between non-adjacent pipeline stages. In our experiments we have discovered that FPGAs such as Virtex [17] are routing and register rich and can support most pipeline-able designs. The number of bits of data computed in each stage is typically less than or equal to the number of logic cells utilized. Hence, the stage to stage communication has enough routing resources by using nearest neighbor interconnect. Extra routing and logic resources (for buffering and multiplexing) have to be utilized for data values communicated across non-adjacent pipeline stages. In the partitioning algorithm, the remaining area in a partition is reduced to reflect the buffering requirements.

A limitation of our approach is that partitions might have bad memory performance when the computation is highly irregular or there are a large number of data dependencies in the DAG. The approximation of routing resources results in infeasible designs in some cases. But, for most applications, the circuits were finally mapped within the available logic and routing resources.

3.5 Phase 2: Pipeline Segmentation

The configuration of the pipeline is generated from the partitions that are computed in Phase 1 by the algorithm in Figure 2. Each partition is utilized to generate one segment of the pipeline. The goal in the segmentation phase is to generate permutations of the pipeline stages in each segment to reduce the

reconfiguration costs across segments. We use the heuristic of matching the corresponding stages of the different pipeline segments. In each partition, the nodes of the same height have the flexibility of being mapped in any order onto the pipeline. In addition, once a node has been mapped onto the pipeline, its successors from the same partition can also be mapped.

The algorithm proceeds by first identifying the list of tasks from each partition that are *Ready* to be scheduled. A task node is *Ready* if all of its predecessors have already been scheduled onto the segment. At the next step, a maximal matching set of task nodes are identified from the set of all *Ready* lists from all *Partitions*. A maximal matching set corresponds to the task node which occurs in most partitions. This step schedules similar nodes from different partitions onto the different segments. This enables the reduction in the reconfigurations costs at runtime. The *Ready* lists are updated before scheduling the next set of nodes. The resulting pipeline schedules with the different segments are shown in Table 1(b) and Table 2(b).

Segment 1	A	M	M	A	A
Segment 2	M	A	M	A	A
Segment 3	S	A	M	*	*

Segment 1	A	M	M	A	A
Segment 2	A	M	M	A	A
Segment 3	A	S	M	*	*

Table 1. Schedules for N-body simulation (a) S_0: Greedy Scheduling (b) S_I: Schedule after Phase 2

Segment 1	M	M	M	*	*
Segment 2	M	S	A	A	A
Segment 3	S	S	*	*	*

Segment 1	M	M	S	A	S
Segment 2	M	M	A	A	S

Table 2. Schedules for FFT (a) S_0: Greedy Scheduling (b) S_I: Schedule after Phase 2

3.6 Reconfiguration of *null* stages

Reconfiguring from a *null* stage to a computation stage can be accomplished by small modifications to the pipeline design. The data values from the previous computation stage are also communicated directly to the output register in addition to flowing through the computational units. 2-input multiplexers are utilized at the output registers to latch one of the two values. Run Time Reconfiguration using partial reconfiguration only needs to modify the SRAM bits controlling the configuration of the multiplexers. This reconfiguration cost is significantly lower than reconfiguring the whole datapath.

4   Results

We evaluate the performance of our techniques by comparing them with a greedy heuristic based on list scheduling. The greedy schedule chooses the largest available *Ready* node as the next stage of the pipeline. A new pipeline segment is

```
 1: Function Segmentation(G, Partition)
 2: ∀v_i : Mapped(v_i) ← FALSE
 3: Num_Partitions ← |Partition|
 4: repeat
 5:    for i = 1 to i = Num_Partitions do
 6:      Ready[i] ← {v_j|v_j ∈ Partition[i] and
 7:                      ∀v_k : v_k = Predecessor(v_j) and Mapped(v_k)}
 8:    endfor
 9:    for i = 1 to i = Num_Partitions do
10:      for all v_j ∈ Ready[i] do
11:          Count(v_j) ← ∑_{l=1}^{Num_Partitions} |{v_k|Type(v_k) = Type(v_j) and
12:                                                   v_k ∈ Ready[l]}|
13:      end for
14:    end for
15:    V_curr ← null
15:    for i = 1 to i = Num_Partitions do
16:      v_sel = v_j | v_j ∈ Ready[i] and ∀v_j :  max{Count(v_j)} and v_j ∈ V_curr
17:      if v_sel = null then
18:          v_sel = v_j | v_j ∈ Ready[i] and ∀v_j :  max{Count(v_j)}
19:      end if
20:      Segment[i] ← Segment[i] ⊕ v_sel
21:      if v_sel != null then
22:          V_curr ← V_curr ∪ v_sel
23:          Mapped(v_sel) ← TRUE
24:      end if
25:    end for
26: until (∀i : empty(Partition[i]))
```

Fig. 2. Algorithm to generate the pipeline segments

initiated when no more nodes can be added to the current segment. The resulting schedule is shown in Table 1(a) and Table 2(a). We utilize the modules and the parameters from the Virtex component libraries [17]. Some of the modules utilized are tabulated below in Table 3. The number of pipelined stages, precision of the inputs and the size of the module when mapped onto the device are listed in the table.

For the N-body simulation and FFT examples, the number of slices to be reconfigured for each schedule is shown in Table 4. This is the reconfiguration cost R_{loop} as defined in Section 3.1. The heuristic based algorithms have a significant saving in the reconfiguration cost. This translates to a direct reduction in the total execution time of the configuration. In the worst case, our heuristic algorithms generate a schedule which is at least as good as the greedy heuristic.

The total execution cost was computed for both the applications for a data set size of 4096 data points with the an on-chip memory size of 2KB (M). For the two example applications, reconfiguration cost is the dominant cost in

8

Module	Stages	Input	Slices	Speed
Add	1	16x16	10	173 MHz
Add	1	32x32	20	157 MHz
Subtract	1	16x16	11	141 MHz
Shift	1	16x16	10	180 MHz
Multiply	1	8x8	39	65 MHz
Multiply	4	8x8	48	131 MHz
Multiply	5	12x12	107	117 MHz
Multiply	5	16x16	168	115 MHz

Table 3. Virtex module characteristics

	Greedy	Our Approach	Speedup
N-body	624	228	2.74
FFT	702	110	6.38

Table 4. Reconfiguration costs in number of Virtex slices

the execution of the application and constitutes more than 95% of the total execution time. The application speedups are of the same order as the speedups in the reconfiguration costs illustrated in Table 4. This shows that our heuristic based approach performs significantly better than the greedy heuristic.

5   Conclusions

Automatic mapping and scheduling of applications is necessary for achieving performance improvement for general purpose computing applications on reconfigurable hardware. These techniques have to address the overheads involved in reconfiguring the hardware. In current architectures the reconfiguration overheads are still significant compared to the execution cost. In this paper, we have proposed algorithmic techniques for mapping and scheduling loops in applications onto reconfigurable hardware. The heuristics we have developed attempt to minimize the reconfiguration overheads by exploiting pipelined designs with partial and runtime reconfiguration. The mapping of example loops from applications illustrates that the proposed algorithms can generate high performance pipelined configurations with reduced reconfiguration cost.

In future work, we will explore the interaction of the proposed techniques with other techniques such as parallelization and vectorization. Reconfigurable hardware specific optimizations such as clock disabling for some pipeline stages and runtime modification of the interconnection to reduce the reconfiguration cost are also being examined.

The work reported here is part of the USC MAARCII project [9]. This project is developing novel mapping techniques to exploit dynamic and self reconfiguration to facilitate run-time mapping using configurable computing devices and architectures. Moreover, a domain-specific mapping approach is being developed to support instance-dependent mapping. Finally, the concept of "active" libraries is exploited to realize a framework for automatic dynamic reconfiguration [8, 13].

References

1. K. Bondalapati. *Modeling and Mapping for Dynamically Reconfigurable Architectures*. PhD thesis, University of Southern California. Under Preparation.
2. K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Reconfigurable Architectures Workshop, RAW'99*, April 1999.
3. K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.
4. K. Bondalapati and V.K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.
5. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.
6. S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
7. D. Chang and M. Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable fpgas. In *IEEE Transactions on Computers*, June 1999.
8. A. Dandalis, A. Mei, and V. K. Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *Reconfigurable Architectures Workshop*, April 1999.
9. MAARCII Homepage. http://maarcII.usc.edu.
10. R. Kress, R.W. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, Sept 1997.
11. W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung. Pipeline Morphing and Virtual Pipelines. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
12. K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. In *IEEE Transactions on Computers*, June 1999.
13. R. P. Sidhu, A. Mei, and V. K. Prasanna. Genetic programming using self-reconfigurable fpgas. In *International Workshop on Field Programmable Logic and Applications*, September 1999.
14. R. Subramanian, N. Ramasubramanian, and S. Pande. Automatic analysis of loops to exploit operator parallelism on reconfigurable systems. In *Languages and Compilers for Parallel Computing*, August 1998.
15. M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures Workshop(RAW' 97)*. ITpress Verlag, April 1997.
16. M. Weinhardt and W. Luk. Pipeline vectorization for reconfigurable systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM '99)*, April 1999.
17. Xilinx Inc.(www.xilinx.com). *Virtex Series FPGAs*.

This article was processed using the LaTeX macro package with LLNCS style