Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices*

Andreas Dandalis, Alessandro Mei**, and Viktor K. Prasanna

University of Southern California
{dandalis, prasanna, amei}@halcyon.usc.edu
http://maarc.usc.edu/

Abstract. Conventional mapping approaches to Reconfigurable Computing (RC) utilize CAD tools to perform the technology mapping of a high-level design. In comparison with the execution time on the hardware, extensive amount of time is spent for compilation by the CAD tools. However, the long compilation time is not always considered when evaluating the time performance of RC solutions. In this paper, we propose a domain specific mapping approach for solving graph problems. The key idea is to alleviate the intervention of the CAD tools at mapping time. High-level designs are synthesized with respect to the specific domain and are adapted to the input graph instance at run-time. The domain is defined by the algorithm and the reconfigurable target. The proposed approach leads to predictable RC solutions with superior time performance. The time performance metric includes both the mapping time and the execution time. For example, in the case of the single-source shortest path problem, the estimated run-time speed-up is 10^6 compared with the state-of-the-art. In comparison with software implementations, the estimated run-time speed-up is asymptotically 3.75 and can be improved by further optimization of the hardware design or improvement of the configuration time.

1 Introduction

Reconfigurable Computing (RC) solutions have shown superior execution times for several application domains (e.g. signal & image processing, genetic algorithms, graph algorithms, cryptography), compared with software and DSP based approaches. However, an efficient RC solution must achieve not only minimal execution time, but also minimal time for mapping onto the hardware [5, 7].

Conventional mapping approaches to RC (see Fig. 1) utilize CAD tools to generate hardware designs optimized with respect to execution time and area.

* This research was performed as part of the MAARC project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

** A. Mei is with the Department of Mathematics of the University of Trento, Italy. This work was performed while he was visiting the University of Southern California.

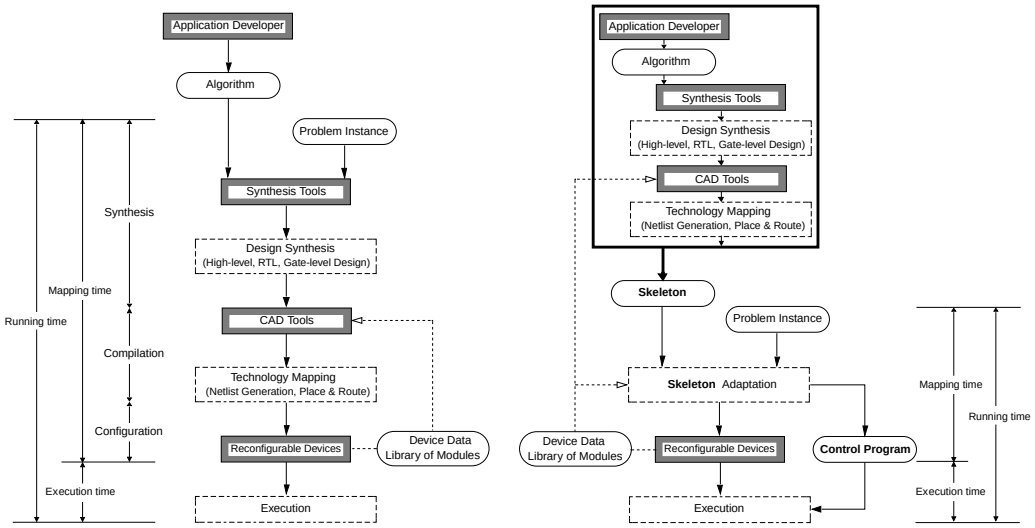

Fig. 1. Conventional (left) and Domain Specific (right) Mapping Approaches

The resulting mappings incur several overheads due to the dominant role of the CAD tools. The clock rate and the required area of the RC solution depend heavily on the CAD tools used and cannot be estimated reliably at compile time. Moreover, the technology mapping phase requires extensive compilation time. Usually, several hours of compilation time is required to achieve execution time in the range of $msec$ [1, 7]. In the case of mappings that are reused over time, compilation occurs only once and is not a performance bottleneck. But, for mappings that depend on the input problem instance, the mapping time cannot be ignored and often becomes a serious performance limitation.

In this paper we propose a novel RC mapping approach for solving graph problems. For each input graph instance, a new mapping is derived. The objective is to derive RC solutions with superior time performance. The time performance metric is the running time which is defined as the sum of the mapping time and the execution time. The key idea of the proposed approach is to reduce the intervention of the CAD tools at mapping time. A technology dependent design is synthesized based on the specific domain [algorithm,target]. The reconfigurable target is now visible to the application developer (see Fig. 1). At run-time, the derived design is adapted to the input graph instance. The proposed mapping approach eliminates the dominant role of the CAD tools and leads to "real-time" RC solutions. Furthermore, the time performance and the area requirements can be accurately estimated before compilation. This is particularly important in run-time environments where the parameters of the problem are not known *a priori* but time and area constraints must be satisfied.

In Section 2, the proposed mapping approach is briefly described. In Section 3, to illustrate our approach, we demonstrate a solution for the single-source shortest path problem. Finally, in Section 4, concluding remarks are made.

2 Domain Specific Mapping

A simple and natural model is assumed for application development. It consists of a host processor, an array of FPGAs, and external memory. The FPGAs are organized as a 2D-mesh. The memory stores the configuration data to be downloaded to the array and the data required during execution. The role of the memory is analogous to the role of the cache memory in a memory system in terms of providing a high-speed link between the host and the array. To illustrate our ideas, we consider an adaptive logic board from Virtual Computer Corporation to map our designs. This board is based on the XC6200 architecture.

In this paper, we consider graph problems as the application domain. Each graph instance leads to a different mapping. Thus, the mapping overhead cannot be ignored. Given a specific domain [algorithm,target], the objective is to derive a working implementation with superior time performance. The time performance metric is the running time which is defined as the sum of the mapping time and the execution time (see Fig. 1). To obtain the hardware implementation, an algorithm specific *skeleton* is synthesized based on the specific domain and is dynamically adapted to the input graph instance at run-time. The proposed mapping approach consists of three major steps (see Fig. 1):

1. **Skeleton design** For a given graph problem, a general structure (*skeleton*) is derived based on the characteristics of the specific domain. The *skeleton* consists of modules that correspond to elementary features of the graph (i.e. graph vertex). The modules are optimized hardware designs and their functionality is determined by the algorithm. Configurations for the modules and their interconnection are derived based on the target architecture.
   The interconnection of the modules is fixed and is defined to be general enough to capture the individual connectivity of different graph instances. Hence, the placement and routing of the modules are less optimized than in conventional CAD tools based approaches. The *skeleton* is derived before compilation and its derivation does not affect the running time. In addition, the *skeleton* exploits low-level hardware details of the reconfigurable target in terms of logic, placement, and routing.
2. **Adaptation to graph input instance** Functional and structural modifications are performed to the *skeleton* at run-time. Such modifications are dictated by the characteristics of the problem instance based on which the configuration of the final layout is derived.
   The functional modifications dynamically add or alter module logic to adapt the modules to the input data precision and problem size. The structural modifications shape the interconnection of the skeleton based on the characteristics of the problem instance.
   A software program (Control Program) is also derived to manage the execution in the FPGAs. This program schedules the operations and the on-chip data flow based on the computational requirements of the problem instance. In addition, it coordinates the data flow to/from the hardware implementation. Since the interconnection of the *skeleton* is well established in Step

1, the execution scheduling essentially corresponds to a software routing for the adapted *skeleton*.

3. **Configuration** Finally, the reconfigurable target is configured based on the adapted structure derived in Step 2. After the completion of the configuration, the control program is executed on the host to initiate and manage the execution on the hardware.

The proposed approach leads to RC solutions with superior time performance compared with conventional mapping approaches. Furthermore, in our approach, the *skeleton* mainly determines the clock rate and the area requirements. Hence, reliable time and area estimates are possible before compilation.

3 The Single-Source Shortest Path problem

To illustrate our ideas, we demonstrate a mapping scheme for the single-source shortest path problem. It is a classical combinatorial problem that arises in many optimization problems (e.g. problems of heuristic search, deterministic optimal control problems, data routing within a computer communication network) [2]. Given a weighted, directed graph and a source vertex, the problem is to find a shortest path from the source to every other vertex.

3.1 The Bellman-Ford Algorithm

For solving the single-source shortest path problem, we consider the Bellman-Ford algorithm. Figure 2 shows the pseudocode of the algorithm [3]. The edge weights can be negative. The complexity of the algorithm is $O(ne)$, where n is the number of vertices and e is the number of edges.

The Bellman-Ford algorithm

```
Initialize G (V,E)
  FOR each vertex  i ε V
    DO  label(i) ← ∞
  label(source) ← 0
Relax edges
  FOR k = 1.. n-1
    DO FOR each edge (i,j) ε E
      DO label(j) ←min {label(j), label(i) + w(i,j)}
Check for negative-weight cycles
  FOR each edge (i,j) ε E
    DO IF label(j) > label(i) + w(i,j)
      THEN return FALSE
  return TRUE
```

Problem size # vertices x # edges	# of iterations m* (average)
16 x 64	4.52
16 x 128	4.40
16 x 240	4.31
64 x 256	4.13
64 x 512	4.06
64 x 1024	4.03
128 x 512	5.32
128 x 1024	4.96
128 x 2048	4.97
256 x 1024	6.04
256 x 2048	5.75
256 x 4096	5.86
512 x 2048	7.02
512 x 4096	6.71
512 x 8192	6.76
1024 x 4096	7.40
1024 x 8192	7.77
1024 x 16384	7.80

Fig. 2. The Bellman-Ford Algorithm and experimental results for m^*

For graphs with no negative-weight cycles reachable from the source, the algorithm may converge in less than $n-1$ iterations [2]. The number of required iterations m^*, is the height of the shortest path tree of the input graph. This

height is equal to the maximum number of edges in a shortest path from the source. In the worst case $m^* = n - 1$, where n is the number of the vertices.

We performed extensive software simulations to determine the relation between m^* and $n-1$ for graphs with no negative-weight cycles. Note that known RC solutions [1] always perform $n-1$ iterations of the algorithm, regardless the value of m^*. Figure 2 shows the experimental results for different problem sizes. For each problem size, $10^5 - 10^6$ graph instances were randomly generated. Then, the value of m^* for each graph instance was found and the average over all graph instances was calculated. For the considered problem sizes, the number of required iterations grows logarithmically as the number of vertices increases. For values of e/n smaller than those in the table in Fig. 2, m^* starts converging to $n-1$.

3.2 Mapping the Bellman-Ford algorithm

The skeleton The *skeleton* corresponds to a general graph G(V,E) with n vertices and e edges. A weight $w(i,j)$ is assigned for each edge $(i,j) \in$ E (i.e. edge from vertex i to vertex j). The derived structure (see Fig. 3) consists of n modules connected in a pipelined fashion. An index $id = 0, 1, .., n-1$ and a *label* are uniquely associated with each module. Module i corresponds to vertex i. The weight of the edges is stored in the memory. No particular ordering of the weights is required. Each memory word consists of the weight $w(i,j)$ and the associated indices i and j.

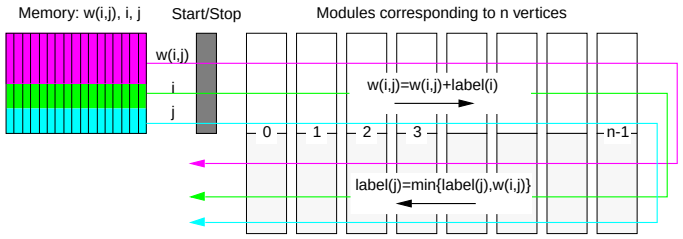

Fig. 3. The *skeleton* architecture for the Bellman-Ford algorithm

The Start/Stop module initiates execution on the hardware. An iteration corresponds to the e cycles needed to feed once the contents of the memory to the modules. The weights $w(i,j)$ are repetitively fed to the modules every e cycles. The algorithm terminates after m^* iterations. One extra iteration is required for the Start/Stop module to detect this termination. If no labels are modified during an iteration and $m^* \leq n$, the graph contains no negative-weight cycles reachable from the source and a solution exists. Otherwise, the graph contains a negative-weight cycle reachable from the source and no solution exists.

In each module (see Fig. 4), the values *id* and *label* are stored and the relaxation of the corresponding edges is performed. In the upper part, the *label* is

added to each incoming weight $w(i,j)$. The index i is compared with id to determine if the edge (i,j) is incident from vertex id. The weight $w(i,j)$ is updated only if $i = id$. In the lower part, the weight $w(i,j)$ is relaxed according to the min operation of the algorithm as shown in Figure 2. The index j is compared with id to determine if the edge (i,j) is incident to the vertex id. The $label$ of the vertex id is updated only if $j = id$ and $w(i,j) < label$. When $label$ is updated, a flag U is asserted.

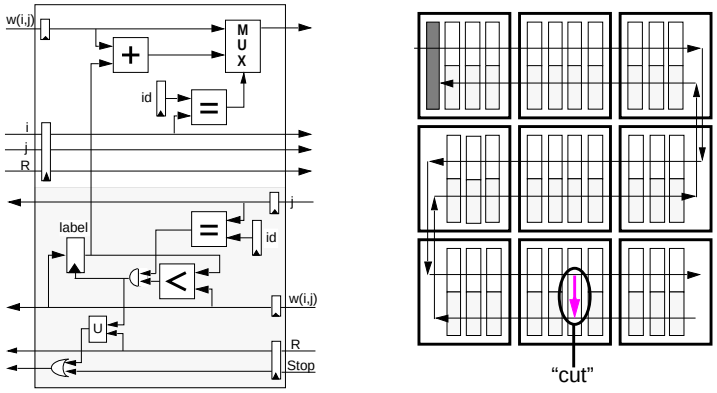

Fig. 4. The structure of the modules (left) and the placement of the *skeleton* into the FPGA array (right)

At the beginning of each iteration, the signal R is set to 1 by the Start/Stop module to reset all the flags. In addition, R resets a register that contains the signal $Stop$ in module $n-1$. The signal $Stop$ travels through the modules and samples all the flags. At the end of each iteration, the $Stop$ signal is sampled by the Start/Stop module. If $Stop = 0$ and $m^* \leq n$, the execution terminates and a solution exists. Otherwise, if $Stop = 1$ after n iterations, no solution exists.

The *skeleton* placement and routing onto the FPGAs array (see Fig. 4) is simple and regular. The communication between consecutive modules is uniform and differs only at the boundaries of the array. Depending on the number of the required modules, a "cut" (Fig. 4) is formed that corresponds to the communication links of the last module (Fig. 3). During the adaptation of the *skeleton* the "cut" is formed and the labels in the allocated modules are initialized. The execution is managed by a control program executed on the host. This program controls the memory for feeding the required weights to the array. In addition, it initiates and terminates the execution via the Start/Stop module.

Area and running time estimates The above module was created based on the parametrized libraries for Xilinx 6200 series of FPGAs [9]. The footprint of each module was $(p + \lceil \log n \rceil) \times (4p + 2\lceil \log n \rceil + 10)$, where p denotes the number of bits in each weight/label, and n is the number of vertices. For $p = \log n = 16$, 4 modules can be placed in the largest device of the XC6200 family. The memory

space required was $(p + 2\lceil \log n \rceil) \times e$ bits, where e is the number of edges. The needed memory-array bandwidth is $p + 2 \log n$ bits/cycle to support the execution. To fully utilize the benefits of the FastMAP^{TM} interface, 140 MB/sec bandwidth is required. Under this assumption, the largest XC6200 device can be configured in 165 μsec using wildcards [8].

The algorithm terminates after $(m^* + 1) \times e + 2n$ cycles, where m^* is the number of required iterations for a given graph. One cycle corresponds to the clock period of the *skeleton*. The clock rate for the *skeleton* was estimated to be at least 15 MHz for p=16 bits, and at least 25 MHz for p=8 bits. In the clock rate analysis, all the overheads caused by the routing were considered. The clock rate was determined mainly by the carry-chain adder of the modules. By using a faster adder, improvements in the clock rate are possible. The mapping time was in the range of *msec*. The above mapping time analysis is based on the timings for the FastMAP^{TM} interface in the Xilinx 6200 series of FPGAs databook [8].

3.3 Performance Comparison

In [1], the shortest path problem is solved by using Dynamic Computation Structures (DCSs). The key characteristic of the solution is the mapping of each edge onto a physical wire. The experiments considered only problems with an average out-degree of 4 and a maximum in-degree of 8. For the instances considered, the compilation time was 4-16 hours assuming that a network of 10 workstations was available. Extensive time was spent for placement and routing. Hence, the resulting mapping time eliminated any gains achieved by fast execution time. To make fair comparisons with our solution, we assumed that the available bandwidth for configuring the array is 4 MB/sec as in [1]. Even though, the mapping time for our solution was estimated to be in the *msec* range (see Fig. 5).

	Check for negative-weight cycles	Mapping time	Execution time			Area requirements
			# of iterations	# of cycles	Clock rate	
Solution in [1]	NO	> 4 hours +	n-1	n-1	$\Omega(1/n^2)$	$\Omega(n^4)$
Our Solution	YES	~ 100 msec ++	m*	(m*+1)e+2n	independent of n	O(n+e)

\+ a network of 10 workstations was used

++ memory-array bandwidth 4MB/sec is assumed as in [1]

Fig. 5. Performance comparisons with the solution in [1]

Besides the mapping overhead, the mapping of edges into physical wires resulted in several limitations in [1], with respect to the clock rate and the area requirements. The clock rate depended on the longest wire which is $\Omega(n^2)$ in the worst case, where n is the number of vertices. This remark is supported by well-known theoretical results [6] which show that in the worst case, a graph takes $\Omega(n^4)$ area to be laid out, where n is the number of vertices, and that the longest wire is $\Omega(n^2)$ long. Therefore, as n increases, the execution time of their solution drops dramatically. For $n = 16, 64, 128$, the execution time was on the

average 1.5-2 times faster than our approach while it became 1.3 times slower for $n = 256$. For larger n, the degradation of performance in [1] is expected to be more severe. Considering both the execution and the mapping time, the resulting speed-up comparing with the solution in [1] was 10^6.

Also, in [1], $n-1$ iterations were always executed and negative-weight cycles could not be detected. If checking for algorithm convergence and negative-weight cycles were included in the design, the resulting longest wire would increase further drastically affecting the clock rate and the excution time. Finally, the time performance and the area requirements in [1] are determined completely by the efficiency of the CAD tools and no reliable estimates can be made before compilation.

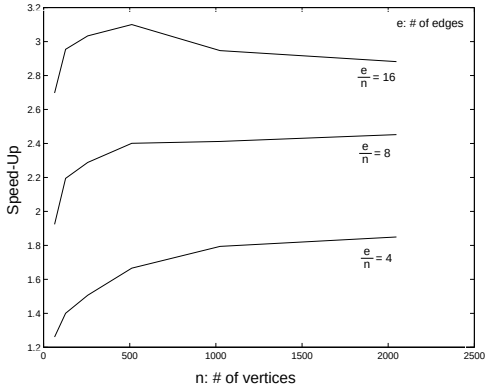

Fig. 6. Comparison of running time: our approach v.s. software implementation.

Area comparisons are difficult to make since different FPGAs were used in [1]. Furthermore, the considered graph instances in [1] were not indicative of the entire problem space since $e/n = 4$. For the considered instances in [1], one XC4013 FPGA was allocated per vertex but, as e/n increases, the area required grows rapidly. In our solution, $O(n)$ area for FPGAs and $O(e)$ memory were required. Moreover, our design is a modular design and can be easily adapted to different graph instances without complete redesign.

Software simulations were also performed to make time performance comparisons with uniprocessor-based solutions. The algorithm that was mapped onto the hardware was also implemented in C language. The software experiments were performed on a Sun ULTRA 1 with 64 MB of memory and a clock rate of 143 MHz. No limitations on the in/out-degree of the vertices were assumed. For each problem instance, $10^4 - 10^6$ graph instances were randomly generated and the average running time was calculated. The compilation time on the uniprocessor to obtain the executable was not considered in the comparisons. Moreover, the data were assumed to be in the memory before execution and no cache effects were considered. Under these assumptions, on the average, an edge was relaxed every 250 $nsec$.

For the hardware implementation, it was assumed that p=16 bits. The mapping time was proportional to the number of vertices of the input graph. Both the mapping and the execution time were considered in the comparisons. The achieved run-time speed-up was asymptotically 3.75. However, for the considered problem sizes (see Fig. 6), lower speed-up was observed. As e/n increases, the mapping time overhead is amortized over the corresponding execution time. Hence, shorter configuration time would result in convergence to the speed-up bound (3.75) for smaller e/n and n.

4 Conclusions

In this paper, a *domain specific* mapping approach was introduced to solve graph problems on FPGAs. Such problems depend on the input graph instance and constitute a suitable application domain to exploit reconfigurability. The proposed approach reduces the dominant role of CAD tools and leads to RC solutions with superior time performance. For example, for the single-source shortest path problem, a speed-up of 10^6 was shown compared with [1]. In comparison with software solutions, an asymptotic speed-up of 3.75 was also shown.

Future work includes more graph problems examples to further validate our mapping approach. In addition, we will focus on specific instances of NP-hard problems where the execution time is comparable to the corresponding mapping time provided by general purpose CAD tools based approaches. We believe that the proposed approach combined with a software/hardware co-design framework can efficiently attack specific instances of NP-hard problems.

References

1. J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures", *SPIE '96: High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, 1996.
2. D. P. Bertsekas, J. N. Tsitsiklis, "Parallel and Distributed Computations: Numerical Methods", Athena Scientific, Belmont, Massachusetts, 1997.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms", The Massachusetts Institute of Technology, 1990.
4. B. L. Hutchings, "Exploiting Reconfigurability Through Domain-Specific Systems", *Int. Workshop on Field Programmable Logic and Applications*, Sep. 1997.
5. A. Rashid, J. Leonard, and W. H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.
6. J. D. Ullman, "Computational Aspects of VLSI", Computer Science Press, Rockville, Maryland, 1984.
7. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.
8. The XC6200 Field Programmable Gate Arrays Databook, http://www.xilinx.com/apps/6200.htm, April 1997.
9. Parameterized Library for XC6200, H.O.T. Works Ver. 1.1, Aug. 1997.