Efficient Datapath Merging for Partially Reconfigurable Architectures

Nahri Moreano, Edson Borin, Cid de Souza, and Guido Araujo

***Abstract*—Reconfigurable systems have been shown to achieve significant performance speedup through architectures that map the most time-consuming application kernel modules or inner loops to a reconfigurable datapath. As each portion of the application starts to execute, the system partially reconfigures the datapath so as to perform the corresponding computation. The reconfigurable datapath should have as few and simple hardware blocks and interconnections as possible, in order to reduce its cost, area, and reconfiguration overhead. To achieve that, hardware blocks and interconnections should be reused as much as possible across the application. We represent each piece of the application as a data-flow graph (DFG). The DFG merging process identifies similarities among the DFGs, and produces a single datapath that can be dynamically reconfigured and has a minimum area cost, when considering both hardware blocks and interconnections. In this paper we present a novel technique for the DFG merge problem, and we evaluate it using programs from the MediaBench benchmark. Our algorithm execution time approaches the fastest previous solution to this problem and produces datapaths with an average area reduction of 20%. When compared to the best known area solution, our approach produces datapaths with area costs equivalent to (and in many cases better than) it, while achieving impressive speedups.**

***Index Terms*—High-level synthesis, reconfigurable computing, resource sharing.**

I. Introduction

It is well known that embedded systems must meet high-throughput, low power, and low cost constraints, specially when designed for signal processing and multimedia applications [1]. These requirements lead to the design of application specific components, ranging from specialized functional units and coprocessors to entire application specific processors. Such components are designed to exploit the peculiarities of the application domain in order to achieve the required performance and to meet the design constraints.

With the advent of reconfigurable systems, new architectural alternatives for the design of complex digital systems have been investigated [2], [3]. The claim of reconfigurable architectures is to provide some degree of software flexibility, while approaching the performance of hardware [4], [5]. Recent work in reconfigurable computing research has shown that a significant performance speedup can be achieved through architectures that map the most time-consuming application kernel modules or inner loops to a reconfigurable datapath [6]–[8]. As a result, it became possible to design application specific components, like specialized datapaths, that can be adapted to perform a different computation, according to the specific part of the application that is running. At run-time, as each portion of the application starts to execute, the system partially reconfigures the datapath so as to perform the corresponding computation.

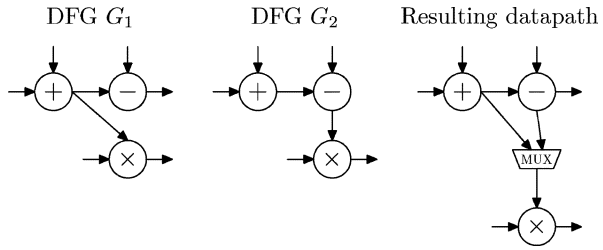

Fig. 1. Data-flow graph merging.

A reconfigurable datapath should have as few and simple hardware blocks (functional units and registers) and interconnections (multiplexers and wires) as possible, in order to reduce its cost, area, and possibly power consumption. To achieve that, hardware blocks and interconnections should be reused across the application as much as possible. Resource sharing has also a crucial impact in reducing the system reconfiguration overhead, both in time and space.

To design such a reconfigurable datapath, one must represent each selected piece of the application as a data-flow graph (DFG) and merge them together, synthesizing a single reconfigurable datapath. The data-flow graph merging process enables the reuse of hardware blocks and interconnections by identifying similarities among the DFGs, and produces a single datapath that can be partially reconfigured at run-time to work for each DFG.

Fig. 1 illustrates the concept of data-flow graph merging. When DFGs G_1 and G_2 from Fig. 1 are merged, the resulting datapath is produced. Notice that in the resulting datapath there are interconnections originated from only one DFG (e.g., the $(+, \times)$ interconnection from DFG G_1) and interconnections shared by both DFGs (e.g., the $(+, -)$ interconnection).

With the advent of new manufacturing technologies, integrated circuits with more and more transistors are being designed, and design productivity has not been able to keep up with transistor density growth [9], [10]. Moreover, interconnection networks are becoming very complex, contributing to

Manuscript received December 19, 2003; revised March 29, 2004, and July 23, 2004. A preliminary version of this work was presented at the International Symposium on System Synthesis, October 2002, Kyoto, Japan. This work was supported in part by research and fellowship grants CAPES 0073/01-6, FAPESP 02/08139-3, 2003/09925-5, and 2000/15083-9, CNPq/CT-Info 55.2117/2002-1, and CNPq 302588/02-7, 478818/03-3, and 301731/2003-9. This paper was recommended by Associate Editor A. Raghunathan.

N. Moreano is with the Federal University of Mato Grosso do Sul, 79070-900 Campo Grande, MS, Brazil (e-mail: moreano@dct.ufms.br).

E. Borin, C. de Souza, and G. Araujo are with the Institute of Computing, University of Campinas, 13084-971 Campinas, Brazil (e-mail: borin@ic.unicamp.br; cid@ic.unicamp.br; guido@ic.unicamp.br).

Digital Object Identifier 10.1109/TCAD.2005.850844

a large share of the total circuit area. As a result, reconfigurable architectures based on coarse-grained logic blocks (instead of fine-grained ones) should be able to improve the usage of computing blocks, while reducing the wiring overhead, due to their high-level organization. It is also necessary the use of efficient allocation techniques which are able to reach better results concerning the component total area (and not only estimating interconnection area) in a reasonable time.

The resource sharing problem is well known in traditional High-Level Synthesis (HLS). There, a datapath is synthesized from a given design behavioral specification, represented by a DFG, optimizing a certain goal (e.g., the hardware cost). The synthesis process consists in the major tasks of scheduling and allocation, where allocation can be further divided into two main subtasks: unit selection and unit binding [11], [12].

Contrary to this problem, in the DFG merge problem studied in this paper, the synthesized reconfigurable datapath must be able to perform the computation of several input behavioral specifications (represented by several DFGs), multiplexed in time, while in HLS the resulting datapath corresponds to only one input DFG. The merging problem is mainly focused in providing inter-DFG resource sharing, while HLS allocation enables intra-DFG resource sharing. In traditional HLS, functional unit and storage binding are performed using estimated interconnection costs, and interconnection binding is done afterward, when the requirements on the interconnections become clear. This is not a good approach to solve datapath merging, because the interconnection area resulting from interconnection binding is highly dependent on functional unit and storage binding.

The main contribution of this paper is a novel and faster technique for the DFG merge problem, with the goal of synthesizing a partially reconfigurable datapath that can be adapted to execute different parts of an application. This technique solves together unit selection and unit binding, minimizing the total area cost of the datapath. Our graph-based approach is built on the solution of a maximum weight clique problem and merges two DFGs at a time. Contrary to most previous solutions, which are based on hardware block mapping, our approach simultaneously maps hardware block and interconnections to compute the reconfigurable datapath. Experimental results using the MediaBench benchmark suite [13], reveal that this technique produces datapaths with area costs equivalent to (and in many cases better than) the best previous (exponential time) solution for area, in much shorter execution times.

This paper is organized as follows. In the next section we describe our datapath architecture model. Section III presents the DFG merge problem more formally and exposes its difficulty. Section IV describes previous work in the literature related to this problem. We present in Section V our graph-based approach to the problem. In Section VI a set of experiments is described to support the efficiency of the proposed approach. Finally, Section VII concludes the work.

II. Architecture Model

The datapath architecture model used in this paper consists of a set of functional units (FUs) and registers (RGs) organized around an interconnection network forming a programmable datapath, as shown in Fig. 2. The interconnection network is based on a set of multiplexers (MUXes) that select the input data for functional units and registers. The datapath is built based on a ASIC (application specific integrated circuit) way and its composition in terms of hardware blocks and interconnections is customized statically toward the computational intensive pieces of the application. That is, the datapath has fixed hardwired logic blocks and only the interconnection network is programmable. Each hardware block is selected from a component library containing from single-operation FUs, such as adders and multipliers, to multifunctional units, such as arithmetic-logic units capable of performing addition, subtraction, and logic operations. Area and resource constraints can be imposed during the construction of the datapath. The reconfigurable datapath is attached to a host processor that executes the remaining of the application. The selected pieces of the application are replaced by new instructions which are executed in the customized datapath.

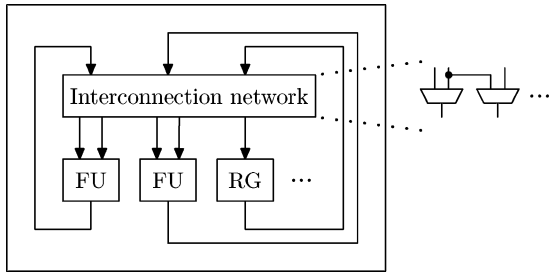

Fig. 2. Architecture model of the reconfigurable datapath.

As the computation progresses, the system partially reconfigures the datapath (setting the MUXes of the interconnection network), such that the pieces of the application are mapped onto it. Given the coarse granularity of the logic blocks (FUs and RGs), fewer bits are needed to reconfigure the datapath (than in the case of fine-grained architectures), thus diminishing the size of the memory required to store the reconfiguration bits (the so called reconfiguration *context*). This is a central issue in SoC (System-on-a-Chip) designs where on-chip area is a premium asset.

III. Data-Flow Graph Merge Problem

In this section we formulate the DFG merge problem more formally. We want to merge several DFGs (corresponding to application portions), in order to build a reconfigurable datapath that is capable of performing the computation of each portion, multiplexed in time, and has the minimum area cost of hardware blocks (functional units and registers) and interconnections. Each application portion i, $i = 1 \ldots n$ is modeled as a DFG G_i, as defined below.

Definition 1: A data-flow graph (DFG) is a directed graph $G = (V, E)$, where:

- A vertex $v \in V$ represents an operation or a variable. Each vertex v has a set of input ports $p = 1 \ldots n_v$ and attributes specifying its type and width (in bits).
- An arc $e = (u, v, p) \in E$ indicates a data transfer from vertex u to the input port p of vertex v.

Given a vertex of a DFG, there may be several hardware blocks (in the component library) where it can be executed.

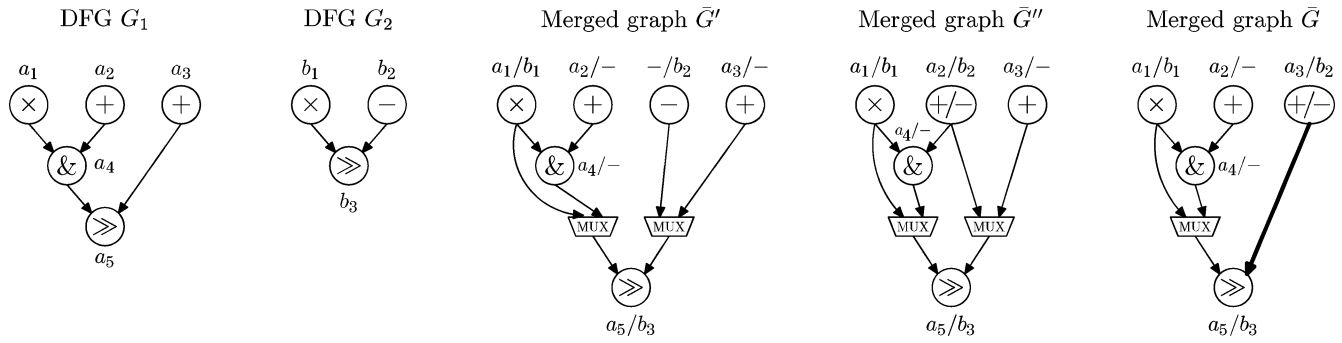

Fig. 3. DFGs G_1 and G_2 and three different merged graphs $\bar{G}'$, $\bar{G}''$, and $\bar{G}$: $A_b(\bar{G}') > A_b(\bar{G}'')$ and $A_b(\bar{G}'') = A_b(\bar{G})$, but $A_i(\bar{G}'') > A_i(\bar{G})$.

Definition 2: The set of hardware blocks $B(v)$ of a DFG vertex v contains the hardware blocks from the component library which can perform the computation represented by v.

The resulting reconfigurable datapath is the merge of all DFGs G_i, $i = 1 \ldots n$ and is modeled as a merged graph $\bar{G}$, as defined below. The merged graph $\bar{G}$ is the overlapping of all G_i, such that only vertices which can be implemented by the same hardware block can be overlapped.

Definition 3: A merged graph, corresponding to DFGs G_i, $i = 1 \ldots n$, is a directed graph $\bar{G} = (\bar{V}, \bar{E})$, where:

- A vertex $\bar{v} \in \bar{V}$ represents a mapping of $m_{\bar{v}}$ vertices v_i, $1 \leq m_{\bar{v}} \leq n$, each one from a different V_i, such that $\bigcap_{\forall v_i} B(v_i) \neq \emptyset$.
- An arc $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$ represents a mapping of $m_{\bar{e}}$ arcs $e_i = (u_i, v_i, p_i)$, $1 \leq m_{\bar{e}} \leq n$, each one from a different E_i, such that all u_i have been mapped onto $\bar{u}$, all v_i have been mapped onto $\bar{v}$, and all p_i match.[1]

The reconfigurable datapath will have a hardware block corresponding to each $\bar{v} \in \bar{V}$, capable of performing all the operations v_i mapped onto $\bar{v}$. For each $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$, there will be a "path" (in the reconfigurable datapath) from the output of $\bar{u}$ to the input port $\bar{p}$ of $\bar{v}$. Moreover, for each input port $\bar{p}$ of each vertex $\bar{v}$ which has more than one incoming arc $(*, \bar{v}, \bar{p})$, the reconfigurable datapath will have a MUX selecting the input operand.

Given a set of n input DFGs G_i, it is possible to build several solutions for the merged graph $\bar{G}$. The optimal solution for $\bar{G}$ is the one which produces the reconfigurable datapath with minimum area cost, considering both hardware blocks and interconnections.

Since in our architecture model the interconnection network is based on MUXes, the interconnection area cost is proportional to the number of MUX inputs. For each arc $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$ which is a mapping of $m_{\bar{e}}$ arcs e_i from DFGs G_i, the MUX (if exists) at the input port $\bar{p}$ of hardware block $\bar{v}$ has $m_{\bar{e}} - 1$ fewer inputs than it would have if no arcs were overlapped. Regarding each MUX as a tree of 2-input MUXes, there is a linear dependency between the number of 2-input MUXes and the number of wires, so the interconnection area cost can be expressed in terms of the number of wires.

The area cost of the reconfigurable datapath is defined below.

[1]The meaning of matching input ports will be further elaborated in Section III-A.

Definition 1: The total area cost $A(\bar{G})$ of the reconfigurable datapath corresponding to the merged graph $\bar{G}$ is

$$A(\bar{G}) = A_b(\bar{G}) + A_i(\bar{G})$$

where $A_b(\bar{G}) = \sum_{\forall \bar{v} \in \bar{V}} A_b(\bar{v})$ and $A_i(\bar{G}) = |\bar{E}| \times A_{\text{mux}}$ are the hardware block and interconnection area cost, respectively, of the reconfigurable datapath. $A_b(\bar{v})$ is the area cost of the hardware block allocated to $\bar{v}$, and A_{mux} represents the area cost equivalent to one MUX input of the suitable width.

We can now define the DFG merge problem, as follows.

Definition 5: Given n input DFGs G_i, $i = 1 \ldots n$, find the merged graph $\bar{G}$, such that $A(\bar{G})$ is minimum.

Finding a mapping of the vertices from the DFGs so as to minimize the hardware block area cost is not a difficult task. It can be modeled as a maximum weight bipartite matching, for which a polynomial time algorithm is well known (the Hungarian Method described in [14]). On the other hand, mapping the arcs from the DFGs so as to minimize the interconnection area is a hard problem (more precisely, it is NP-complete), because the mapping of arcs depends on the mapping of their adjacent vertices. That is, two arcs from two DFGs can only be mapped if their source vertices are mapped as well as their destination vertices. So, if we map vertices without considering the interconnection costs or using only estimates, we may get a solution where the interconnection area cost is not minimized and consequently, the total area cost is also nonoptimal.

We now illustrate these concepts with an example. Given the DFGs G_1 and G_2 in Fig. 3, we can build three different merged graphs $\bar{G}'$, $\bar{G}''$, and $\bar{G}$. In $\bar{G}'$, vertices a_2 from G_1 and b_2 from G_2 are not mapped, so the reconfigurable datapath corresponding to $\bar{G}'$ would have six hardware blocks (a multiplier, two adders, a subtractor, an *and*, and a shifter). In $\bar{G}''$, those vertices are mapped (represented by the notation a_2/b_2), resulting in a reconfigurable datapath with five hardware blocks (a multiplier, an adder/subtractor, an adder, an *and*, and a shifter). So, $A_b(\bar{G}')$ is larger than $A_b(\bar{G}'')$ and consequently, $A(\bar{G}')$ is also larger than $A(\bar{G}'')$.

Still in Fig. 3, $\bar{G}''$ and $\bar{G}$ represent different vertex mappings. In $\bar{G}''$ vertex b_2 of G_2 is mapped onto vertex a_2 of G_1, while it is mapped onto a_3 in $\bar{G}$. The vertex mappings represented by $\bar{G}''$ and $\bar{G}$ may appear equivalent and, as a matter of fact, $A_b(\bar{G}'')$ is equal to $A_b(\bar{G})$. But they allow for different arc mappings. In $\bar{G}''$, no arcs are overlapped, so two MUXes are needed at the two input ports of vertex a_5/b_3. In $\bar{G}$, the arcs (a_3, a_5) and

(b_2, b_3) are mapped (highlighted in the figure), thus eliminating the need for one of the MUXes. As a result, $A_i(\bar{G}'')$ is larger than $A_i(\bar{G})$ and consequently, $A(\bar{G}'')$ is also larger than $A(\bar{G})$.

In order to compute the optimal solution for $\bar{G}$ we have to find out which vertex mapping, among several possibilities, gives the best arc mapping, i.e., which mapping minimizes the total area cost.

We proved that the DFG merge problem is NP-complete [15] by reducing the subgraph isomorphism problem [16] to it.

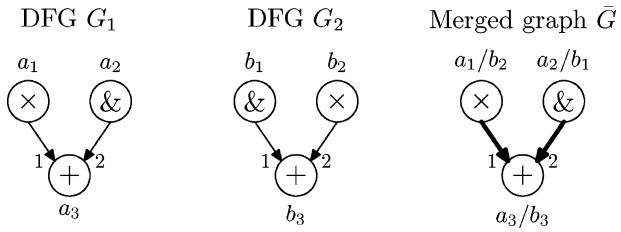

Fig. 4. Interconnection sharing through commutativity.

A. Input Ports and Commutativity

Several two-input operations are commutative, so properly exchanging the sources of these operations can enable arc mappings that would not exist otherwise, thus eliminating wires and MUXes and reducing the interconnection area cost of the reconfigurable datapath. Each operation has a set of input ports which represent the input operands it expects, for instance an addition has two input ports p_1 and p_2. From Definition 3, two arcs $(u_i, v_i, p_i) \in G_i$ and $(u_j, v_j, p_j) \in G_j$ can be mapped if u_i and u_j can be mapped, as well as v_i and v_j, and p_i and p_j match. The input ports p_i and p_j match if: (a) they are equal; or (b) v_i and/or v_j represent two-input commutative operations.

Consider for example the DFGs G_1 and G_2 of Fig. 4, where the input ports 1 and 2 of the addition operations are shown. When merging G_1 and G_2, there are three possible vertex mappings and the two possible arc mappings $(a_1, a_3, 1)/(b_2, b_3, 2)$ and $(a_2, a_3, 2)/(b_1, b_3, 1)$. These arc mappings would not be obtained without exploiting the commutativity of the addition operation, because the input ports of the mapped arcs are different. As a result, the merged graph $\bar{G}$ has two less interconnections and MUXes than it would have without commutativity.

IV. Related Work

In this section we describe the work available in the literature, related to the data-flow graph merge problem.

The most commonly used method to the merge problem combines two DFGs at a time and is based on the problem of finding a maximum weight matching of a bipartite graph. The two partitions of vertices in the bipartite graph correspond to the vertices of the two DFGs being merged. There is an edge connecting two vertices in the bipartite graph if they can be mapped (overlapped). The weight of this edge represents the gain achieved if the two vertices are mapped, concerning the desired objective function. Since each vertex of one DFG should be mapped to at most one vertex of the other, the maximum weight matching of the bipartite graph gives a solution to the problem. This method is a polynomial-time approach, but since vertices are mapped using only estimates about the interconnection mapping, the solutions produced can be far from optimal.

The bipartite matching method has been used in [17]–[19]. In [17] two solutions are presented to synthesize an application specific unit capable of executing in different modes, each one corresponding to one cluster of operations from the application nonscheduled flow graph. The presented solutions solve both unit selection and unit binding and try to minimize the area cost of the resulting unit. The weight of an edge represents the area gain achieved if the two vertices are mapped. This gain includes hardware block and estimated interconnection area. The interconnection area gain, corresponding to the mapping of vertices u and v, is an upper bound estimate, because it considers that the predecessors of u and v in the DFGs being merged have been mapped to each other, as well as the successors (what may not happen). The second solution uses a slightly different approach to handle situations in which the two DFGs being merged exhibit a high degree of similarity and regularity. The relative positions of the vertices in the DFGs are used to break ties when several edges in the bipartite graph have the same weight.

In [18] the authors combine two or more designs into a reconfigurable one, based on the identification and mapping of components common to these designs. The goal is to minimize the reconfiguration time. The bipartite graph edge weights are defined using as criteria the component types, the component positions in the FPGA, and the depth from matched ports. In [19] this technique is used to design a reconfigurable datapath capable of performing the computation corresponding to time-consuming inner loops of an application. The bipartite graph edge weights represent estimates on the interconnection mapping.

Another group of methods [17], [20] relies on search mechanisms to explore the solution space looking for the optimal solution. Since the solution space has an exponential size and is explored randomly, these methods execution times may be not polynomially bounded, although in practice they sometimes behave reasonably well.

In [20] a merging technique is proposed to synthesize a multifunctional processing unit, in which each function corresponds to clusters of operations in the specification. The method assigns operations to the processing unit's operators, given a number of pre-allocated operators. The approach is based on iterative improvement and simulated annealing local search algorithms, and aims at reducing the interconnection cost of the processing unit. Both algorithms start with an initial random assignment, and continuously step through a randomly chosen neighborhood solution set, considering a cost acceptance criterion. The concept of neighborhood is defined using a two-exchange mechanism, where the assignment of two operations from the same DFG are exchanged. The neighborhood of the current solution is the set of solutions that can be reached from it by performing only one two-exchange. The solution provided by the iterative improvement algorithm is the first local minimum found, so it may be far from optimal. The simulated annealing method accepts limited deteriorating transitions, trying to escape from local minima, and as a consequence has longer execution times.

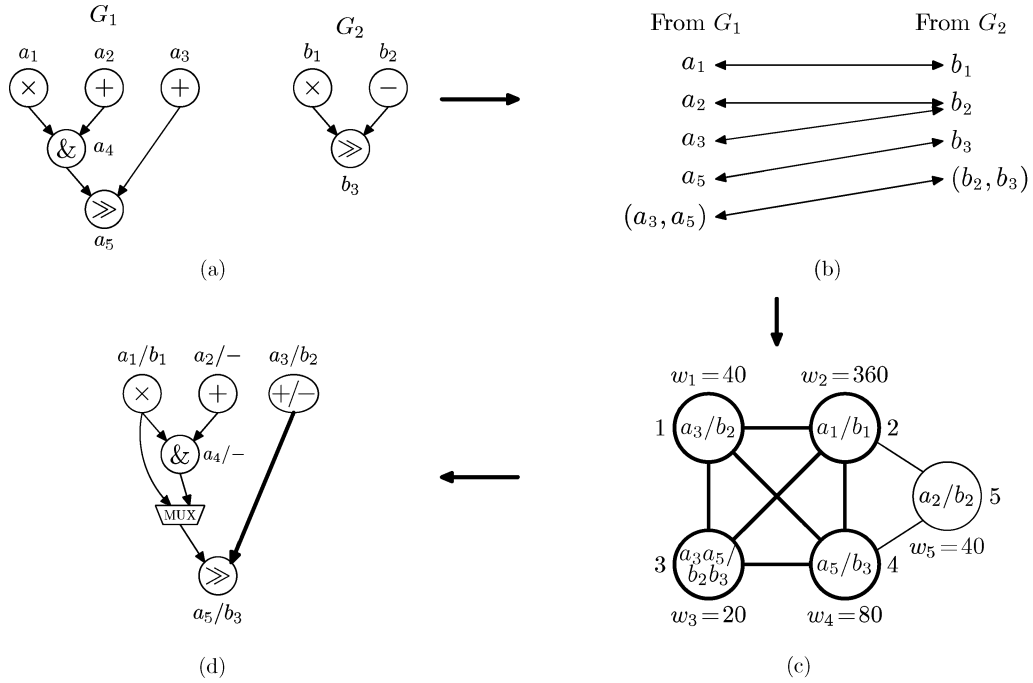

Fig. 5. Steps of DFG merging algorithm. (a) DFGs G_1 and G_2; (b) all possible mappings of vertices and arcs of G_1 and G_2; (c) maximum weight clique on the compatibility graph G_c; (d) merged graph $\tilde{G}$.

A solution described in [17] also uses an iterative improvement method based on the algorithm presented in [20], but performs unit selection and unit binding, and starts with a different initial solution. Again the solution space is explored by stepping from one solution to one of its neighbors, reached by a two-exchange. The cost acceptance criteria does not permit deteriorating transitions, but zero-cost difference transitions are accepted.

In another solution from [17], which also merges two DFGs at a time, an ILP (Integer Linear Programming) model is developed to represent a global interconnection-cost model. In this case, the interconnection area gain corresponding to the mapping of the vertices u and v is dependent on the mapping of its predecessors in the DFGs. This model provides an exact but exponential-time solution to the problem of merging only two clusters, and can be used as an approximation in order to merge several DFGs iteratively.

None of the above referred solutions exploit operation commutativity in order to improve the interconnection mapping.

A merging approach is proposed in [21] to design a reconfigurable datapath capable of performing the computation corresponding to time-consuming inner loops of an application. The authors use a graph-based technique to perform hardware block and interconnection assignment together, trying to minimize the interconnection area cost. Our merge solution is an extension of this approach, but it is more complete and precise because we model both hardware block and interconnection area costs and we globally solve unit selection and unit binding. Our algorithm solves both unit selection and binding, while the referred solution solves only unit binding and assumes that the blocks have been previously allocated. Besides, our technique tries to minimize the total area of the datapath (including hardware blocks and interconnections), while the referred solution tries to minimize only the interconnection area. Finally, our algorithm exploits the commutativity of the operations in order to increase the interconnection sharing, while the referred solution does not.

V. Hardware Block and Interconnection Mapping Approach

In this section, we propose a technique to solve the DFG merge problem. Our approach solves the unit selection and unit binding tasks, and performs functional-unit, storage, and interconnection binding simultaneously. So, we are able to find a global solution and optimize the interconnection network using accurate area costs instead of estimates, even for the case in which the interconnection network is MUX-based. Our method merges two DFGs at a time and in order to merge several DFGs, this method is applied iteratively. First, two input DFGs are merged, then the resulting graph is merged with another input DFG, and so on.

A. Compatibility Graph

Initially, a compatibility graph is constructed to represent all possible vertex and arc mappings between two input DFGs G_i and G_j and the consistency among these mappings. Two vertices from G_i and G_j respectively, can be mapped (overlapped) if there is, in the component library, a module capable of performing the operations they represent. Two arcs from G_i and G_j can be mapped if their corresponding source vertices can be mapped, as well as their destination vertices, and their input

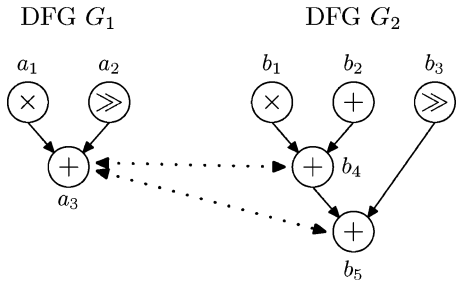

Fig. 6. Incompatible mappings: a_3 maps to b_4 and b_5.

ports match.[2] Given the conditions for matching input ports defined in Section III-A, the compatibility graph also contains arc mappings obtained exploiting the commutativity of operations. Fig. 5(b) lists those vertices and arcs from DFGs G_1 and G_2 in Fig. 5(a) that can be overlapped to each other. Each mapping is represented by a double-arrow line connecting the vertices or arcs that can be overlapped. We formally define the compatibility graph below.

Definition 6: A compatibility graph, corresponding to DFGs G_i and G_j, is an undirected weighted graph $G_c = (V_c, E_c)$, where:

- Each vertex $v_c \in V_c$ with weight w_c corresponds to:
  - — a possible mapping v_i/v_j of vertices $v_i \in G_i$ and $v_j \in G_j$ such that $B(v_i) \cap B(v_j) \neq \emptyset$; or
  - — a possible mapping $(u_i, v_i, p_i)/(u_j, v_j, p_j)$ of arcs $(u_i, v_i, p_i) \in G_i$ and $(u_j, v_j, p_j) \in G_j$ such that $B(u_i) \cap B(u_j) \neq \emptyset$, $B(v_i) \cap B(v_j) \neq \emptyset$, and p_i and p_j match.
- There is an edge $e_c = (u_c, v_c) \in E_c$ if the mappings represented by u_c and v_c are compatible.

In order to build the compatibility graph we need also to define the notion of *mapping compatibility*.

Definition 7: Two vertex or arc mappings are not compatible whenever they map the same vertex of G_i to two different vertices of G_j, or vice-versa.

This compatibility criterion is illustrated in Fig. 6. In that figure two DFGs G_1 and G_2 are shown. There are two possible arc mappings between G_1 and G_2, which are $(a_1, a_3)/(b_1, b_4)$ and $(a_2, a_3)/(b_3, b_5)$. These two mappings are not compatible since they map the same vertex a_3 from G_1 to two different vertices, b_4 and b_5, in G_2.

The weight of vertices in the compatibility graph are assigned using the concept of area reduction. If we map two vertices v_i and v_j, we will have, in the reconfigurable datapath, only one hardware block capable of computing both v_i and v_j operations, instead of having two hardware blocks to perform each operation, respectively. When we map two arcs, instead of having, in the reconfigurable datapath, two interconnections, we will have only one, and we will not need a MUX at the input port of the destination hardware block. The area reduction achieved by this mapping corresponds to the area cost equivalent to one MUX input of the suitable width.

Definition 8: The weight w_c of vertex $v_c \in V_c$ is the area cost reduction achieved with the mapping represented by v_c, where

- If v_c represents a mapping v_i/v_j, then $w_c = A_b(v_i) + A_b(v_j) - A_b(v_i/v_j)$.
- If v_c represents a mapping $(u_i, v_i, p_i)/(u_j, v_j, p_j)$, then $w_c = A_{\text{mux}}$.

By using the definitions presented above the compatibility graph can be easily constructed. Fig. 5(c) shows the compatibility graph G_c resulting from the mappings of vertices and arcs from G_1 and G_2 in Fig. 5(b). Consider, for example, mappings a_3/b_2 (vertex 1 in G_c) and $(a_3, a_5)/(b_2, b_3)$ (vertex 3 in G_c). For those mappings, no vertex from G_1 maps to two distinct vertices in G_2 and vice-versa. As a result, the two mapping are compatible, and an edge (1,3) is required in G_c. On the other hand, no edge exist in G_c between vertices 3 and 5. The reason is that the mappings represented by 3 and 5 are incompatible, since b_2 in G_2 maps to both a_2 and a_3 in G_1. The vertex weights represent the area cost reductions of the corresponding mappings. For instance, the weight w_1 of the mapping a_3/b_2 corresponds to the area cost reduction achieved by having one adder/subtractor functional unit, instead of one adder and one subtractor.

B. Maximum Weight Clique Solution

In order to determine the resulting graph $\bar{G}$ such that $A(\bar{G})$ is minimum, it is necessary to find the set of mappings that are compatible to each other and provide the maximum area cost reduction. This can be achieved by computing the maximum weight clique of the compatibility graph G_c.

Definition 9: The maximum weight clique of a graph $G_c = (V_c, E_c)$ is a set of vertices $C \subseteq V_c$ where, for all vertices u_c, $v_c \in C$, the edge $(u_c, v_c) \in E_c$, and $\sum_{\forall v_c \in C} w_c$ is maximum.

In the compatibility graph G_c of Fig. 5(c), the maximum weight clique has vertices 1, 2, 3, and 4 (highlighted in the figure). The maximum weight clique problem is also NP-complete [16], thus a heuristic polynomial-time algorithm is used to solve it.

Finally, the mappings represented by the vertices from the maximum weight clique C of G_c are used to construct the optimal merged graph $\bar{G}$. Since the clique gives the mappings which achieve the maximum area cost reduction, the merge graph produced is the one with minimum area cost. Each vertex from C gives a vertex or arc mapping between G_i and G_j that will become a vertex or arc in $\bar{G}$, respectively. If a vertex in C represents an arc mapping obtained through commutativity, the input ports of the corresponding destination vertex of $\bar{G}$ are properly set. Those vertices and arcs from G_i and G_j that do not belong to any mapping in C, are also inserted into $\bar{G}$ as unmapped vertices and arcs. For example, using the clique shown in Fig. 5(c), we get the optimal merged graph $\bar{G}$ shown in Fig. 5(d).

Notice that if the arc mapping $(a_3, a_5)/(b_2, b_3)$ (node 3 of G_c) had not been included into the compatibility graph and only vertex mappings had been considered, the maximum weight clique could have been the set of vertices 2, 4, and 5 of G_c,

[2]Note that the compatibility graph used here has a different meaning from its traditional use in HLS. In HLS, the vertices of the compatibility graph are in one-to-one correspondence with the operations of the input DFG. An edge in the compatibility graph represents two operations that can be bound to the same resource. In our approach, each vertex in the compatibility graph represents a possible mapping of two operations or data transfers from two input DFGs, i.e., two operations or data transfers that can be bound to the same resource. An edge in the compatibility graph represents two mappings that can be used together.

```
MergeDFGs
    input: n DFGs G_i = (V_i, E_i), i = 1...n
    output: merged graph Ḡ = (V̄, Ē),
        such that A(Ḡ) is minimum
    /* Initially, Ḡ is the first input graph G_1 */
    Ḡ ← G_1;
    /* Iteratively merge Ḡ with input graph G_i */
    for i ← 2 to n do
        G_c ← ConstructCompatibilityGraph(Ḡ, G_i);
        C ← FindMaximumWeightClique(G_c);
        Ḡ ← ReconstructResultingGraph(C, Ḡ, G_i);
```

Fig. 7. DFG merge based on hardware block and interconnection mapping.

resulting in the merged graph $\bar{G}''$ shown in Fig. 3, which, as we have already seen, has a larger total area cost than $\bar{G}$. This shows the importance of performing simultaneous vertex and arc mappings.

In this approach, both unit selection and unit binding are solved. For each vertex of $\bar{G}$, one hardware block is allocated and its type is selected based on the operations mapped on the vertex. These operations are bound to this hardware block. For each arc in $\bar{G}$, a connection is required in the reconfigurable datapath, and for each vertex of $\bar{G}$ which has more than one arc coming to one of its input ports, a MUX is allocated. The arc mappings represented in $\bar{G}$ correspond to the interconnection binding of the data transfers.

C. DFG Merging Algorithm

Our DFG merging method is shown in Fig. 7. It computes an exact solution for merging two DFGs, if we use an exact algorithm to the maximum weight clique problem. Since we use a polynomial time heuristic for the clique problem, our algorithm runs in polynomial time. In order to merge two DFGs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the worst-case time complexity of the "ConstructCompatibilityGraph" step is $O(|E_1|^2 \times |E_2|^2)$, since $|V_c|$ is at most $O(|E_1| \times |E_2|)$. The algorithm used in the "FindMaximumWeightClique" step is based on a branch-and-bound technique [22]. We tuned this algorithm such that its execution time is polynomially bounded by $|V_c|$, that is, we interrupt the search when this time limit is reached. Finally, the worst-case time complexity of the "ReconstructResultingGraph" step is $O(|E_1| \times |E_2|)$. This worst-case scenario arises when all vertices of G_1 and G_2 represent the same commutative operation, which in practice is never the case.

D. Exploiting Commutativity

Traditionally, commutativity is exploited in pre- or post-processing steps, so it only uses estimate information about unit binding or it is based on highly constrained alternatives. Our DFG merge algorithm exploits operation commutativity during the binding task, and since hardware block and interconnection bindings are unified, the exchange decisions are taken using precise area costs. During the construction of the compatibility graph, we already consider the possible source exchanges. When we find the maximum weight clique, we choose the mappings that result in the maximum area cost reduction, which may include some mappings obtained by commutativity.

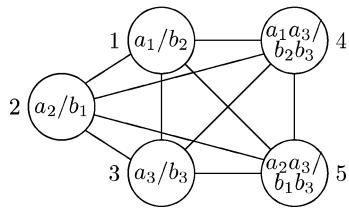

Fig. 8. Compatibility graph G_c with mappings of G_1 and G_2 from Fig. 4 (arc mappings obtained through commutativity).

For example, given the DFGs G_1 and G_2 of Fig. 4, we get the compatibility graph G_c shown in Fig. 8. Notice that the arc mappings represented by vertices 4 and 5 of G_c are obtained only if commutativity can be exploited.

VI. Experimental Results

This section describes a set of experiments we performed in order to evaluate the datapath merging approach and the merging technique proposed in this work. Our solution to the DFG merge problem was applied to a number of programs from the MediaBench suite [13]. Enough experimental evidence exists to support the fact that inner loops account for the largest share of program execution time. Therefore, these loops are good candidates for mapping onto a reconfigurable datapath. Each program was compiled using the GCC compiler, and profiled so as to determine which inner loops contributed the most to the program execution time. For each such loop, a DFG was generated from the loop body RTL code (GCC intermediated representation) [23]. Using RTL instead of machine instructions permitted us to extract the loop code after most machine-independent code optimizations, but before register allocation and machine-dependent optimizations. Moreover, whenever possible, procedure integration (automatic inlining) was applied. The section of application code corresponding to a DFG can contain control constructions, such as "if-then", "if-then-else", and "switch". For simplicity, we do not handle nested loops. The DFGs were generated using a technique based on if-conversion and using condition bit vectors.

We considered up to eight inner loops for each application (if available). The loops/DFGs that required too many resources were discarded, in order to guarantee that the resulting datapath would meet area and resource constraints of the target architecture. In all experiments, the host processor was a SPARC-v8 processor.

For each application, the DFGs were merged iteratively using our merge algorithm, starting from the larger DFG to the smaller one (with respect to the number of vertices). We performed some experiments with different orderings of the input DFGs. Although the decreasing size ordering achieved better results in many cases, no significant differences in area for the different orderings were obtained, as also reported in [17].

A. Comparing Merging Techniques

Experiments were performed in order to evaluate our proposed merging algorithm with respect to other merging techniques. In the experiments described in this subsection, HLS was performed on each DFG beforehand, using an in-house tool.

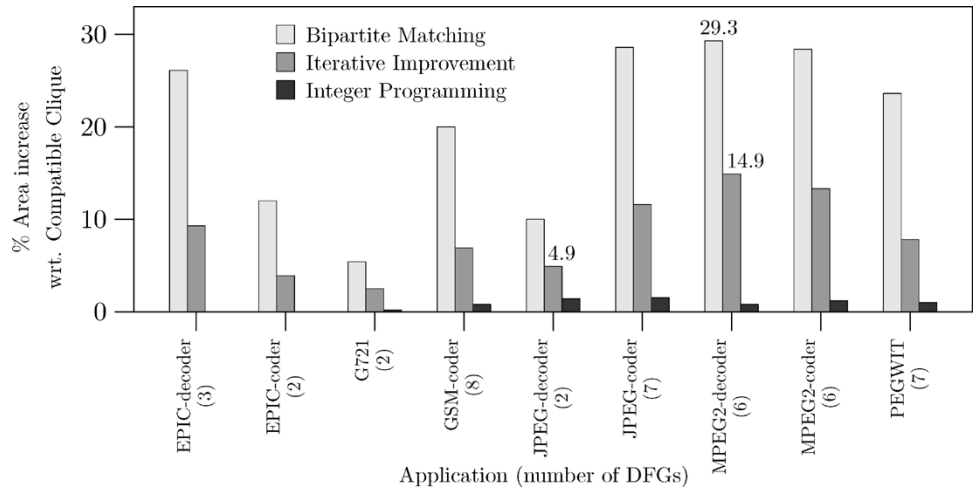

Fig. 9. Area increase of previous methods wrt. our Compatible Clique (the missing bars correspond to area increase of 0%).

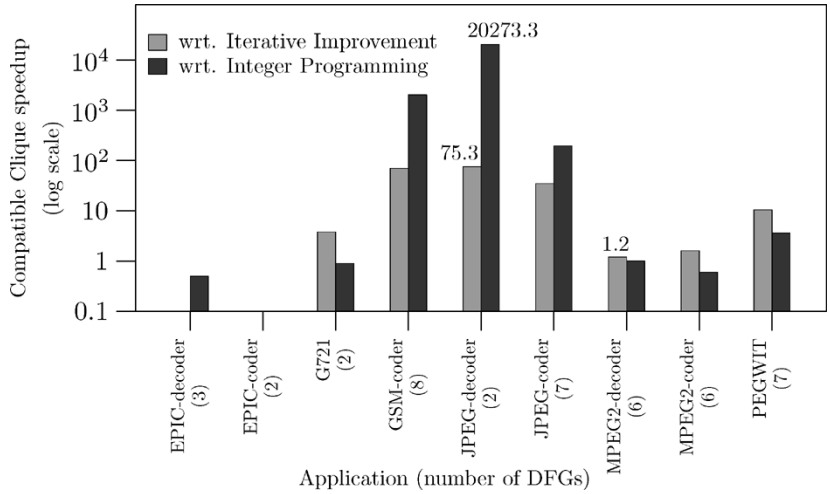

Fig. 10. Our Compatible Clique speedup wrt. previous methods (the missing bars correspond to speedup of 0.1).

Chaining was exploited during scheduling and no intra-DFG resource sharing was exploited. The DFGs resulting from HLS were merged afterward. The area and delay information used were obtained from a component library. The control units of the input DFGs were not unified on the merged datapath, so control logic area was not measured.

We implemented three techniques based on previous approaches to DFG merge and compared their solutions to ours, with respect to both the area of the resulting reconfigurable datapath and the execution time of the algorithm. These methods are the bipartite maximum weight matching approach (in this section referred as Bipartite Matching) [17]–[19], the iterative improvement local search algorithm (referred as Iterative Improvement) [17], [20] and the ILP solution (referred as Integer Programming) [17]. Our approach, based on computing the solution to a compatible maximum weight clique, is referred as Compatible Clique.

In a first experiment we compared, for each application program, the area of the resulting datapath produced by our approach, and by the three previous techniques. Fig. 9 shows the percentage of area increase produced by the previous methods when compared to our approach. We also measured the execution time of each technique. Fig. 10 shows the speedup achieved by our approach when compared to the other methods (note the logarithm scale).

Notice that Bipartite Matching produces datapaths with area up to 30% larger than our algorithm. Iterative Improvement produces area increase from 2.5% to 14.9%, when compared to our Compatible Clique, but it also takes much longer than our approach. For example, for the JPEG decoder program, Iterative Improvement produces an area increase of only 4.9%, but it takes 75.3 times longer to execute. For the MPEG2 decoder program, its execution time is only 20% longer than our Compatible Clique, but the area increase is 14.9%. Integer Programming is almost equivalent to our Compatible Clique concerning the area of the resulting datapath, but its execution time can be up to 20 thousand times longer than the Compatible Clique approach. For some applications, our Compatible Clique produced datapaths with smaller area than even the Integer Programming approach. Although this method provides the exact solution when merging two DFGs, it does not guarantee to find the best solution when merging several DFGs iteratively. Moreover, this method does not exploit operation commutativity, while our approach does.

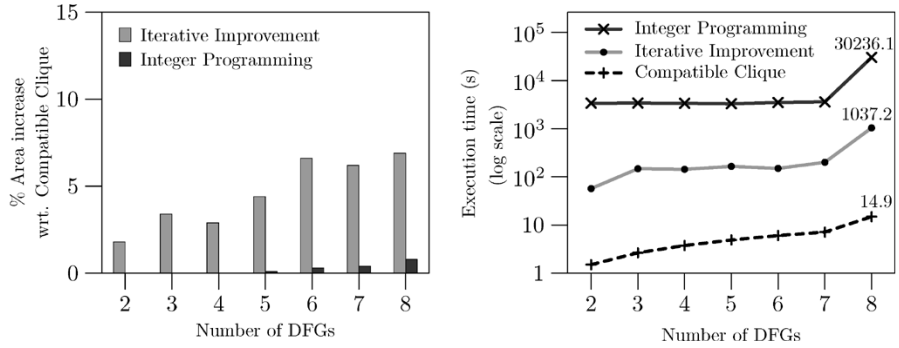

Fig. 11. Area increase wrt. our Compatible Clique and execution time for GSM coder (the missing bars correspond to area increase of 0%).

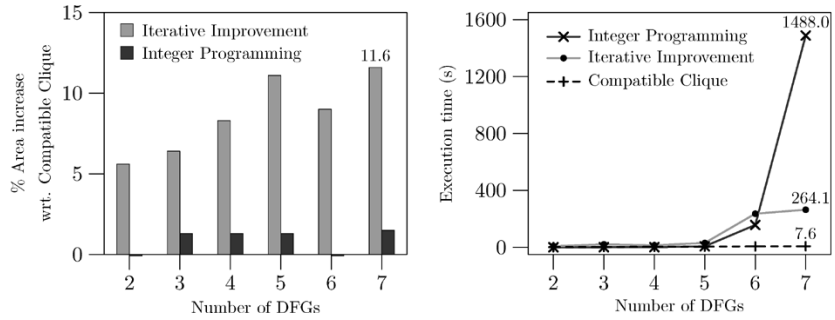

Fig. 12. Area increase wrt. our Compatible Clique and execution time for JPEG coder (the downward bars correspond to area increase of -0.1%).

TABLE I
COMPARISON OF OUR COMPATIBLE CLIQUE SOLUTION AND PREVIOUS APPROACHES

Merge method	Avg area increase wrt. Compatible Clique	Compatible Clique avg speedup
Bipartite Matching	20.4%	0.1
Iterative Improvement	8.4%	21.8
Integer Programming	0.8%	2500.4

Table I summarizes the results of Figs. 9 and 10 and shows the average area increase of the previous methods with respect to our algorithm and the average speedup obtained by our algorithm over the others. The Bipartite Matching approach is very fast (average execution time of 0.6 s, while Compatible Clique average execution time is 4.8 s), but it produces poor results for datapath area (average area increase of 20.4%, when compared to our Compatible Clique approach), so it is not a good choice if we have strict area constraints. Iterative Improvement produced an average area increase of 8.4%, while taking 21.8 times longer than our Compatible Clique. Finally, the datapath areas produced by Integer Programming technique are almost equivalent to (but slightly worse than) our Compatible Clique, but its exponential execution time is prohibitive.

In a second experiment, we analyzed the behavior of the merge techniques when the number of merged DFGs grows. We merged, for each application, the two most relevant DFGs (which correspond to the two loops which contributed the most to the program execution time), followed by the three most relevant DFGs, and so on. Again, for each merge execution, DFGs were merged iteratively from the larger to the smaller one (with respect to the number of vertices). Figs. 11 and 12 show these results for the GSM and JPEG encoding programs,

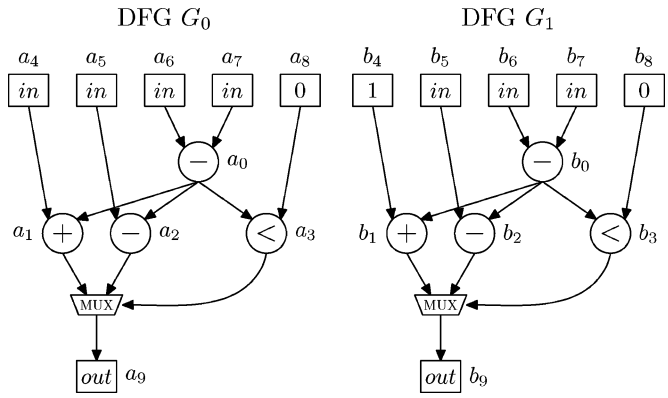

Fig. 13. DFGs G_0 and G_1.

respectively. For both applications we noticed that Integer Programming execution time, as expected, grows exponentially and Iterative Improvements also grows very drastically, while our Compatible Clique execution time grows polynomially.

Hence, our algorithm produces resulting datapaths with area costs equivalent to (and in many cases better than) the best known solution for area (Integer Programming), which takes exponential time. At the same time, our execution time approaches the fastest previous solution (Bipartite Matching), which produces the worst area result.

B. Comparing Datapath Merging versus Datapath Combining

A pertinent question one may pose is if we need the datapath merging approach at all. We could simply combine all input DFGs into a complete DFG, just multiplexing their inputs and outputs. Then we could apply HLS to this complete DFG and let the HLS tool exploit resource sharing. Therefore, we want

```
...
(37)  if id = '0' then -- DFG 0
(38)     t0 := in2v - in3v ;
(39)     t1 := in0v + t0 ;
(40)     t2 := in1v - t0 ;
(41)     if t0 < zero32 then
(42)        t3 := t2 ;
(43)     else
(44)        t3 := t1 ;
(45)     end if ;
(46)     out0 <= t3 ;
(47)  else -- DFG 1
(48)     t0 := in1v - in2v ;
(49)     t1 := one32 + t0 ;
(50)     t2 := in0v - t0 ;
(51)     if t0 < zero32 then
(52)        t3 := t2 ;
(53)     else
(54)        t3 := t1 ;
(55)     end if ;
(56)     out0 <= t3 ;
(57)  end if ;
...
```

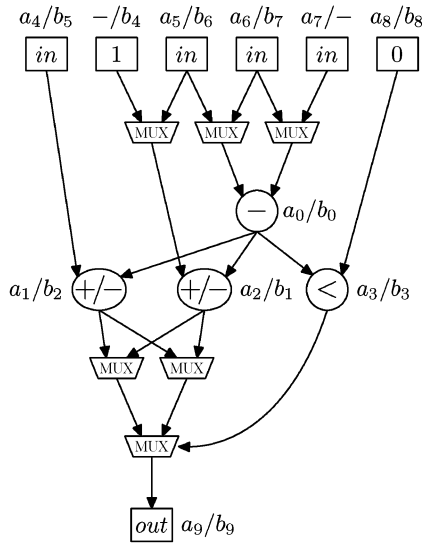

Fig. 14. VHDL code combining G_0 and G_1 and combined datapath at Synopsys DC output.

```
...
(37)  -- Merged DFG
(38)  t0 := in2v - in3v ;
(39)  if id = '0' then
(40)     t1 := in0v + t0 ;
(41)  else
(42)     t1 := one32 + t0 ;
(43)  end if ;
(44)  t2 := in1v - t0 ;
(45)  if t0 < zero32 then
(46)     t3 := t2 ;
(47)  else
(48)     t3 := t1 ;
(49)  end if ;
(50)  out0 <= t3 ;
...
```

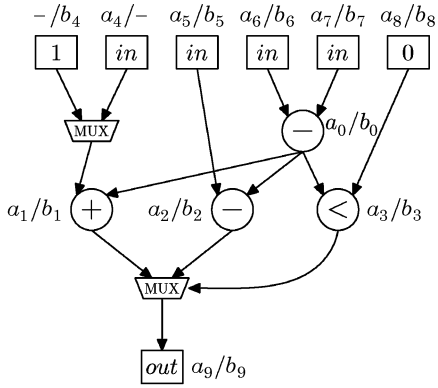

Fig. 15. VHDL code merging G_0 and G_1 and merged datapath at Synopsys DC output.

to know if the datapath merge approach can provide some area gain, when compared to this pure HLS approach. In order to address this issue we performed another set of experiments, which are described in this subsection.

In these experiments, the input DFGs were merged using our algorithm and we generated a datapath for the merged DFG. We also generated a datapath (referred as combined datapath) for a complete DFG combining all input DFGs without merging. Then these two datapaths were synthesized under the same design constraints and optimization options, and the synthesis results were compared.

We used the Synopsys Design Compiler (DC) synthesis tool (version 2003.03) [24], which performs three levels of optimization: architectural optimization, logic-level and gate-level optimization. During architecture optimization, DC exploits common subexpression sharing and resource sharing. We provided design constraints so as to optimize the design to minimize its area (setting the directives *set_resource_allocation* and *set_resource_implementation* to *area_only* and *set_max_area* to 0.0). We also directed the optimization process to apply the maximum effort in the mapping and area reduction phases (specifying both options *-map_effort* and *-area_effort* of the *compile* optimization command set to *high*).

We illustrate this experiment with an example. Given the DFGs G_0 and G_1 in Fig. 13, we produced a combined datapath, as well as, a merged datapath. Figs. 14 and 15 show the datapath and the corresponding VHDL code, for the combined and merged datapaths, respectively. When we submit these two datapaths to DC, the merged datapath provides an area reduction of 27% when compared to the combined datapath.

By analyzing the resource allocation and sharing reports generated by DC during synthesis, shown in Fig. 16, one can see that in the combined datapath, the subtractions a_0 and b_0 of G_0 and G_1 (lines 38 and 48 in Fig. 14, respectively) were mapped onto the same subtractor (resource r23 in Fig. 16(a)). However, the addition a_1 and subtraction b_2 (lines 39 and 50) were mapped onto an adder/subtractor (resource r22), as well as, the subtraction a_2 and addition b_1 (lines 40 and 49) which were mapped onto resource r79. In the merged datapath, there is also a subtractor corresponding to the mapped operation a_0/b_0 (line 38 in Fig. 15 and resource r22 in Fig. 16(b)). But differently from the combined datapath, a_1/b_1 (lines 40 and 42) shared the same adder (resource r70), and a_2/b_2 (line 44) shared a subtractor (resource r73). As a result, the combined datapath uses two adder/subtractors for operations which the merged datapath implements with only one adder and one subtractor. Besides, fewer

Resource	Module	Parameters	Contained Operations
r21	DW01_cmp2	width=32	lt_41/lt/lt
			lt_51/lt/lt
r22	DW01_addsub	width=32	add_39/plus/plus
			sub_50/minus/minus
r23	DW01_sub	width=32	sub_38/minus/minus
			sub_48/minus/minus
r79	DW01_addsub	width=32	add_49/plus/plus
			sub_40/minus/minus

(a)

Resource	Module	Parameters	Contained Operations
r20	DW01_cmp2	width=32	lt_45/lt/lt
r22	DW01_sub	width=32	sub_38/minus/minus
r70	DW01_add	width=32	add_40/plus/plus
			add_42/plus/plus
r73	DW01_sub	width=32	sub_44/minus/minus

(b)

Fig. 16. Resource allocation and sharing reports generated by Synopsys DC synthesis tool. (a) Combined datapath. (b) Merged datapath.

TABLE II
NON-MERGING VS. MERGING APPROACH COMPARISON

Application (number of DFGs)	Combined datapath area	Merged datapath area	Area reduction
EPIC-decoder (3)	15404.0	14634.0	5.0%
EPIC-coder (2)	15760.0	14732.0	6.5%
G721 (2)	2755.0	1144.0	58.5%
GSM-coder (8)	53702.0	49392.0	8.0%
JPEG-decoder (2)	6482.0	5417.0	16.4%
JPEG-coder (7)	21985.0	19432.0	11.6%
MPEG2-decoder (6)	5399.0	4304.0	20.3%
MPEG2-coder (6)	11291.0	8780.0	22.2%
PEGWIT (7)	5280.0	3815.0	27.7%
Average			19.6%

MUXes are needed in the merged datapath, resulting in the area reduction.

We observed that DC is very good in mapping mutually exclusive operations which receive the same operands. But it may introduce additional MUXes when these operations have different operands. The merge algorithm, on the other hand, maps input and output operands, as well as operations, and permits us to map exclusive operations which may have different operands, if it results in the final datapath area reduction. In order to accomplish this, it analyzes hardware block and interconnection mapping together.

Table II shows the area results achieved (measured in DC area units), for the MediaBench applications, using DC synthesis of the combined datapath, as defined above, and the merged datapath produced by our merging technique. The area reduction obtained with the merging approach is also shown. The results strongly indicate that the datapath merging technique indeed provides area reduction, when compared to a nonmerging approach.

VII. CONCLUSION

This paper presented a novel graph-based technique for the DFG merge problem. Performance speedup can be achieved through architectures that map the most time-consuming application kernels and inner loops to a partially reconfigurable datapath. We represented each such fragment as a DFG and merged them together into a single reconfigurable datapath. Our approach merges the individual DFGs into a single reconfigurable datapath one at a time. At each step, it solves a maximum weight clique problem that allocates and binds hardware blocks and interconnections at the same time, minimizing the area cost of the resulting reconfigurable datapath.

Experiments were performed to evaluate the efficiency of the proposed algorithm and to compare it with previous solutions to the DFG merge problem. Area of the resulting reconfigurable datapath and the execution time of the algorithm were measured. Our algorithm and the previous techniques were applied to merge the DFGs corresponding to the most relevant loops from a set of programs in the MediaBench suite.

One of the previous approaches was very fast (average execution time of 0.6 s, while ours is 4.8 s), but it produced poor datapath area results (average area increase of 20.4%, when compared to our solution). Another previous method produced an average area increase of 8.4%, while taking 21.8 times longer than our approach. Finally, the datapath areas produced by the last previous technique are almost equivalent to (but slightly worse than) ours, but its exponential execution time is prohibitive. We conclude that our interconnection mapping algorithm produces resulting datapaths with area costs equivalent to (and in many cases better than) the best known solution for area (which takes exponential time) while at the same time, our execution time approaches the fastest previous solution, which produces the worst area results.

REFERENCES

[1] W. Wolf, *Computers as Components—Principles of Embedded Computing System Design*. San Mateo, CA: Morgan Kaufmann, 2001.

[2] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *Proc. Design Automation Conf.*, 1999, pp. 610–615.

[3] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A quick safari through the reconfiguration jungle," in *Proc. Design Automation Conf.*, 2001, pp. 172–177.

[4] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surveys*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[5] K. Bondalapati and V. Prasanna, "Reconfigurable computing systems," *Proc. IEEE*, vol. 90, no. 7, pp. 1201–1217, Jul. 2002.

[6] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[7] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[8] H. Schmit *et al.*, "PipeRench: a virtualized programmable datapath in 0.18 micron technology," in *Proc. Custom Integrated Circuits Conf.*, 2002, pp. 63–66.

[9] SEMATECH. (2002) International Technology Roadmap for Semiconductors—1999 Edition/2002 Update. [Online]. Available: http://public.itrs.net

[10] F. Vahid, "Making the best of those extra transistors," *IEEE Des. Test Comput.*, vol. 20, no. 1, p. 96, Jan./Feb. 2003.

[11] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis—Introduction to Chip and System Design*. Boston, MA: Kluwer, 1992.

[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communication systems," in *Proc. MICRO-30*, Dec. 1997, pp. 330–335.
[14] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization—Algorithms and Complexity*. New York: Dover, 1998.
[15] N. Moreano, G. Araujo, and C. Souza, "CDFG Merging for Reconfigurable Architectures," Institute of Computing-UNICAMP, Tech. Rep. IC-03-18. [Online]. Available: http://www.ic.unicamp.br/ic-tr-ftp/2003/03-18.ps.gz, 2003.
[16] M. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[17] W. Geurts, F. Catthoor, S. Vernalde, and H. De Man, *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Boston, MA: Kluwer, 1997.
[18] N. Shirazi, W. Luk, and P. Cheung, "Automating production of run-time reconfigurable designs," in *Proc. 6th Symp. FCCM*, Apr. 1998, pp. 147–156.
[19] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks," in *Proc. Design Automation Test Eur. Conf.*, 2001, pp. 735–740.
[20] A. van der Werf *et al.*, "Area optimization of multifunctional processing units," in *Proc. Int. Conf. CAD*, Nov. 1992, pp. 292–299.
[21] N. Moreano, G. Araujo, Z. Huang, and S. Malik, "Datapath merging and interconnection sharing for reconfigurable architectures," in *Proc. Int. Symp. Syst. Synthesis*, Oct. 2002, pp. 38–43.
[22] S. Niskanen and P. Östergärd, "Cliquer User's Guide, Version 1.0," Commun. Lab., Helsinki Univ. Technol., Helsinki, Finland, Tech. Rep. T48, 2003.
[23] R. Stallman. (2002, Jan.) GNU Compiler Collection Internals. [Online]. Available: http://gcc.gnu.org/onlinedocs/
[24] Synopsys, Inc., "Design Compiler User Guide,", Mountain View, CA, 2003.

Nahri Moreano received the B.S. degree in computer science and the M.S. degree in computer engineering from the Federal University of Rio de Janeiro, Brazil, in 1990 and 1994, respectively. She is currently pursuing the Ph.D. degree in computer science at the University of Campinas, Brazil.

Her research interests include computer architecture and compiler optimizations for embedded systems, hardware/software codesign, and reconfigurable computing.

Edson Borin received the B.S. degree in computer science from the Federal University of Mato Grosso do Sul, Brazil, in 2001. He is currently pursuing the Ph.D. degree in computer science at the University of Campinas, Brazil.

His research interests include computer architectures, processor specialization, and compiler optimizations.

Cid de Souza received the B.S. and M.S. degrees in electrical engineering from the Catholic University of Rio de Janeiro, Brazil, in 1985 and 1989, respectively, and the Ph.D. degree in applied sciences from the Catholic University of Louvain, Belgium, in 1993.

He is currently an Assistant Professor of the Institute of Computing at the University of Campinas, Brazil. His research interests include combinatorial optimization, linear and integer programming and the design and analysis of algorithms.

Guido Araujo received the B.S. degree in electrical engineering from UFPE, Brazil, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1994 and 1997, respectively.

He worked as a Compiler Consultant for Conexant Semiconductor Systems and Mindspeed Technologies from 1997 to 1999. He is currently an Associate Professor at the University of Campinas, Brazil. His main research interests are compiler and computer architecture optimization for embedded processors and reconfigurable computing.

Dr. Araujo was corecipient of best paper awards at the 1996 ACM/IEEE Design Automation Conference, and at the 2003 International Workshop on Software and Compilers for Embedded Systems (SCOPES'03). He was also awarded the 2002 University of Campinas Zeferino Vaz Award for his contributions to Computer Science research and teaching.