Profile-Driven Instruction Mapping for Dataflow Architectures

Mongkol Ekpanyapong, Michael Healy, and Sung Kyu Lim

***Abstract*—Dataflow architectures provide an abundance of computing units that can be statically or dynamically configured to match the computing requirements of the given application. Wire delay has a reduced impact in dataflow architectures because only neighboring architectural entities are allowed to communicate within a single clock cycle. In this paper, the authors propose integer linear programming (ILP)-based placement and routing algorithms for mapping dataflow graphs (DFGs) to dataflow machines. The optimization process is guided by profiling information available from the compiler. The goal is to minimize the total execution time of the given application represented by a DFG under architectural constraints. A hierarchical method to handle the complexity of the initial ILP formulation is proposed. The profile-driven ILP algorithm reduces the total execution time of benchmark applications compared to the conventional wirelength-driven ILP approach. In addition, the ILP-based approach outperforms simulated annealing-based approach.**

***Index Terms*—Dataflow architecture, instruction mapping, placement, routing.**

I. Introduction

As wire delay increasingly becomes a significant performance bottleneck in monolithic architectures, with their centralized structure requiring fast communication over long distances, there is a strong motivation to move to dataflow architectures. Dataflow computing is a computing paradigm with an abundance of computing units that can be statically or dynamically configured to match the computing requirements of the given application. Dataflow architectures distribute their arithmetic logic units (ALUs), storage units, and communication paths over a two-dimensional grid and enable enormous parallelism in computation and communication by eliminating complex centralized control. They fire operations into ALUs as soon as the required input operands became available. The results are then routed to other ALUs waiting on them. Wire delay has a reduced impact since only neighboring architectural entities are allowed to communicate within a single clock cycle. This allows dataflow architectures to be extremely scalable compared to the traditional von Neumann architecture.

An integral part of the dataflow computing is application mapping. The dataflow graph (DFG) is used to model the flow of data among the instructions, where each node in DFG is mapped to a unique computing resource in the target dataflow machine. The total number of clock cycles needed to finish executing the given DFG on a target dataflow machine is heavily dependent upon the quality of DFG mapping solution. This is because the communication delay is determined by the distance between the instructions as well as the routing switch delays along the paths. Thus, an intelligent mapper would place frequently and time critically communicating parts of the computation close to each other, thereby delivering very high performance. Thus, the effective mapping of applications to the dataflow architecture grid requires a synergistic interaction between compilers and physical design. A more rigorous approach is to combine compilation and placement. This combined approach allows compilers to expose more information to the placement such as the access profile among different instructions. On the other hand, compilers gain geometric information for the instructions on the dataflow architecture and perform more optimization.

The contribution of this paper is the introduction of the new dataflow mapping problem along with the first placement and routing algorithms. The optimization process is guided by profiling information available from the compiler. Our goal is to minimize the total execution time of the given application represented by a DFG under architectural constraints. Our solution is based on integer linear programming (ILP) formulation. ILP-based approach generally produces better quality results at the cost of more runtime compared to other methods such as greedy heuristics or simulated annealing (SA) [1]. In this paper, we propose a hierarchical method to handle the complexity of the initial ILP formulation. Our profile-driven ILP algorithm reduces the total execution time of benchmark applications compared to the conventional wirelength-driven ILP approach. In addition, our ILP-based approach outperforms SA-based approach.

Our ILP-based instruction mapping for dataflow architecture is similar to ILP-based netlist mapping for field-programmable gate arrays (FPGAs). There exist several works that solve the FPGA problem using ILP-based approach [2]. ILP approach is also used for application-specific integrated circuit (ASIC) designs [3]–[6]. These two problems are similar in that directed graphs are used to represent the program/circuit to be mapped, and the hardware resource constraints are expressed via integer linear equations. Each node in the graph is associated with some computing latency/delay, and the mapping solution determines the overall communication/signal propagation delay of application/circuit mapped to the target hardware. On the other hand, the granularity of the target hardware element is quite different in these two problems, where our dataflow mapping deals with instructions, ALUs, and memories, whereas FPGA mapping deals with logic gates, flip-flops, and lookup tables. Another difference is the mapping objective, where FPGA mapping is mostly concerned with interconnect length, congestion/routability, and longest path delay measured from the "hardware" mapped onto FPGA. On the other hand, the total execution time of given "software" that is mapped to dataflow machine is the major focus during our DFG mapping. Thus, the exploitation of profiling information via program simulation is only possible and meaningful in our dataflow mapping problem.

The rest of this paper is organized as follows: Section II discusses the dataflow computing and related works. Section III presents the problem formulation. Our DFG mapping algorithm is described in Section IV. Experimental results are presented in Section V, and we conclude in Section VI.

II. Preliminaries

A. Dataflow Architectures

Dataflow machines are perhaps the best studied alternative to von Neumann processors. The dataflow model of execution offers attractive properties for parallel processing. First, dataflow computing bases instruction execution on operand availability; thus, synchronization of parallel activities is implicit. Second, dataflow instructions do not constrain sequencing except for data dependencies in the program. Thus, the DFG representation of a program exposes all forms of parallelism, eliminating the need to explicitly manage parallel execution. For high-speed computations, the advantage of the dataflow approach over the control-flow method stems from the inherent parallelism embedded

Manuscript received September 3, 2005; revised February 4, 2006 and April 27, 2006. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) Polymorphous Computer Architecture Program under Contract F33615-03-C-4105 and in part by the National Science Foundation under Contract CNS-0411149. This paper was recommended by Associate Editor K. Bazargan.

The authors are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0250 USA.

Digital Object Identifier 10.1109/TCAD.2006.883927

at the instruction level. This allows efficient exploitation of fine-grain parallelism in application programs. Due to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. The first dataflow architectures [7] appeared in the mid- to late 1970s, and there was a notable revival in the late 1980s and early 1990s [8]–[10]. Currently, there are ongoing research works that extend the dataflow computing to handle nonstreaming applications such as TRIPS [11], WaveScalar [12], and MONARCH [13].[1]

B. Related Works

Among the several existing works that perform instruction scheduling via temporal partitioning and mapping [15]–[17], the work by Nagarajan *et al.* [17] is the closest work to ours. They presented a static instruction scheduler named static placement, dynamic issue (SPDI) for their TRIPS [11] architecture. The SPDI scheduling combines compiler-driven placement of instructions with hardware-determined issue order. In a TRIPS architecture that uses the SPDI scheduling, instructions execute in dataflow order with each instruction issuing when its inputs become available. SPDI-based TRIPS architectures thus retain the benefits of static placement.

There exist several differences between SPDI and our work. First, SPDI assumes that the size of DFG could be larger than that of the target architecture, i.e., more instructions to map than the number of ALUs available. Thus, SPDI first places the instructions to the ALU grid and then schedules them on a timeline so that the hardware handles issuing properly. On the other hand, we assume that our grid-based dataflow architecture can always accommodate a given DFG so that no external instruction scheduling and issuing are necessary—a pure dataflow characteristic. Second, SPDI does not perform routing and thus relies only on the physical location of the ALUs to estimate the access latency among the placed instructions. In our case, routability is an important issue during DFG placement and is handled carefully. A detailed routing information is used not only to configure our dataflow machine but also to estimate access latency more accurately. Last, our DFG mapping is profile guided, where the compiler propagates the simulation results of the target application so that the subsequent DFG mapping process can explicitly target the critical paths identified by the compiler. On the other hand, no profiling is used in SPDI.

III. Problem Formulation

A. Compilation Flow

First, an application is fed into the system by the front-end compiler. The DFG is then generated. High-level machine-independent compiler optimization is performed here. Then, low-level optimization is invoked during back-end compilation. We modify the Trimaran compiler [18] such that the mapper reads the annotated DFG, performs placement, and annotates the placement and routing solution back to the assembler. Then, the assembler is used to generate the binary for a given architecture. Note that the architecture description is read by compiler, mapper, and assembler such that minor architecture modifications can be done without system modification. Statistic information for the given application is extracted from the front-end compiler and available for other parts of flow.

[1]Another architecture that is related to our work is RAW [14]. RAW processor is partitioned into many tiles, where each tile is composed of a small processor. Unlike dataflow-based architecture, the mapping of RAW is not done at instruction level.

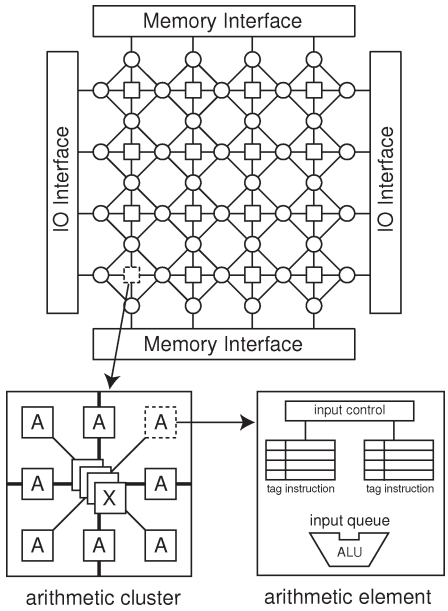

Fig. 1. Description of generic dataflow fabric, where squares represent arithmetic clusters and circles represent memory clusters. Our arithmetic cluster contains eight arithmetic elements and four multiplexing elements. The operand queues in our arithmetic element enable an effective nonstreaming application execution.

B. Architecture and Program Model

Fig. 1 shows an illustration of the dataflow architecture used in this paper.[2] Squares in the dataflow fabric represent arithmetic clusters, and circles represent memory clusters. Fig. 1(c) shows the extension made to the ALUs, which is similar to the ALUs used in WaveScalar [12]. In each ALU, the input operands contain a tag field that must be matched in order for the ALU to execute the instruction. Therefore, this extension requires additional buffers, comparison hardware, and extension of the operand field to include tag assignments. Along with additional hardware, additional instructions are inserted by the compiler to handle this extended feature. A tag generation mechanism is included for the modifications made to the ALUs. The purpose of this tag system is to allow many iterations of a loop to be executed in parallel whenever there are available resources. Throughout our experiment, we assume that each arithmetic block consists of eight ALUs and four multiplexers, whereas memory blocks consist of four memory nodes and four multiplexers.

DFGs are used to identify which instructions produce data needed by other instructions. If-conversion [19] is performed to convert control dependencies in an application into dataflow edges. Operations in DFGs can now be conditionally executed by consuming a predicate operand produced by the original control-branching condition. An instruction is only executed if its predicate input is set to true. Loops in a program are captured as cycles in the DFG. After the compiler constructs a DFG, each node in the DFG is mapped to some processing element on the dataflow architecture grid. General processing elements consist of ALUs and input buffers to store operands. Processing elements can execute and communicate in parallel subject to dataflow constraints captured by the DFG. Typically, dataflow architectures

[2]This architecture is not an imaginary one. Instead, our dataflow mapping tool presented in this paper is targeting MONARCH architecture [13] developed by USC/ISI and Raytheon Corporation that supports dataflow computing mode. In fact, MONARCH is an example of polymorphic computing architecture (PCA) that can dynamically morph into dataflow, RISC, or SIMD mode depending on the computing needs.

have built-in flow-control mechanisms. A processing element producing a result will stall automatically if the input buffers on a processing element consuming the result are full. Similarly, a processing element will not execute until all its inputs are available.

C. DFG Mapping Problem

Our DFG mapping process is divided into placement and routing steps. We model DFG using $G(V, E)$, where V is the set of dataflow nodes and E is the set of dataflow edges. There are three types of dataflow nodes: 1) arithmetic; 2) memory; and 3) multiplexing. Each node is associated with a nonuniform delay, where more complex operations such as multiplication and division incur larger delay than simpler operations such as addition and subtraction. Each edge $e(x, y) \in E$ is associated with "profile" information that denotes how many times x accessed y during a DFG simulation. Section III-D discusses how the simulation is performed.

We model the dataflow architecture using a graph $A(N, W)$, where N is the set of architectural elements and W is the set of architectural wires connecting the elements. Our dataflow architecture contains two levels of hierarchy: 1) element level and 2) cluster level. Each element can accommodate (and thus execute) a single dataflow node, and each cluster contains multiple elements. The basic assumption is that our generic architectural element (i.e., ALU) can be configured to execute any type of arithmetic operations that appeared in a given DFG. Thus, each architectural element is configured to execute the instruction mapped to it. There are three types of architectural elements: 1) arithmetic; 2) memory; and 3) multiplexing. If a DFG node v is mapped to an architectural element n, the type of v has to match that of n. There are two types of architectural clusters: arithmetic and memory clusters. An arithmetic cluster contains a set of arithmetic and multiplexer elements. A memory cluster contains a set of memory and multiplexer elements. The multiplexing elements are used to connect arithmetic and memory elements.

In this paper, we assume that $|V| < |N|$ and $|E| < |W|$, i.e., the size of given DFG is always smaller than the given architecture.[3] Thus, DFG partitioning and/or scheduling [16] is not necessary. Formal problem definitions are given as follows.

Definition 1: DFG placement problem is to map each DFG node $v \in V$ to a unique architecture element $n \in N$ such that the type and capacity constraints are satisfied.

Definition 2: DFG routing problem is to map each DFG edge $e \in E$ to a path in A such that the wire capacity constraint is satisfied. A path consists of all related architectural elements and intercluster/intracluster wires.

Our minimization objective includes wirelength and profiling weight. Wirelength is calculated at two stages of the DFG mapping: during placement and routing. An illustration is shown in Fig. 2. The wirelength of a DFG edge during placement is the Manhattan distance between the source and sink based on a cluster-level grid. We ignore the wirelength difference for intracluster connections. The wirelength of a DFG edge during routing is the number of clusters along the edge. The profiling weight is equal to the normalized access frequency gathered by the front-end compiler. The minimization of profile-weighted wirelength improves the total execution time of the application as evidenced by our related experiments shown in Section V. The estimation of the total execution time is explained in the following section.

[3]It is important to note that a given DFG may still not be mappable (i.e., unroutable) depending on the quality of placement and routing solution as evidenced by our related experiments shown in Section V. Thus, one of our objectives is to improve the routability during clustering and placement.

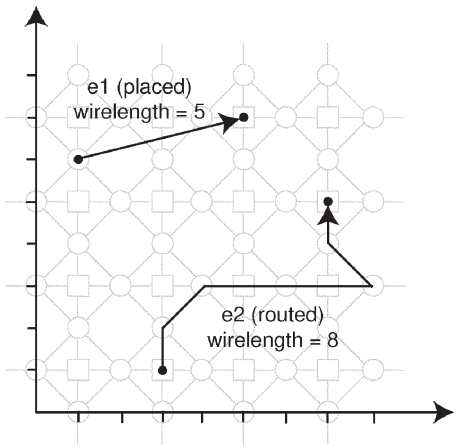

Fig. 2. Wirelength computation during placement $(= e_1)$ and routing $(= e_2)$.

D. Profiling and Execution Time Estimation

During profiling, high-level simulations are performed on C source codes. By running the application on sample input sets, statistic information, such as how many times each path is executed, is collected. This statistic information is then annotated back into each edge in the DFG. In SPEC2000, there are three different input sets available: test, train, and reference. The test input set is just for quickly testing whether the compiler/architecture is working correctly or not. The train input set is used to train compiler/architecture in case profiling is desired. The reference input set is intended to give a complete evaluation of the host computer system's performance. Thus, we use the train input set during our profiling stage and reference input set for reporting our results. We assume perfect memory and large enough input queue buffers. Note that WaveScalar [12] also assumes perfect L1 data cache and unbounded input queues. In addition, TRACE simulation [20] is also similar to this approach.

Since our dataflow computer has no speculation, we estimate the total execution time of a given application as follows[4]: The estimation is based on the number of times each DFG node and edge are executed. For each path $p \in G$, the execution time of p, denoted $\text{exec}(p)$ and measured in clock cycle, is computed as follows:

$$\text{exec}(p) = \sum_{e(u,v) \in p} \{\text{freq}(e) \cdot (\text{delay}(u) + \text{delay}(e))\}.$$

$\text{freq}(e)$ denotes the access frequency collected during the profiling. $\text{delay}(u)$ denotes the computation latency of a DFG node u, which is dependent on the complexity of the ALU operation u performs (i.e., a part of dataflow machine specification). $\text{delay}(e)$ denotes the communication delay incurred during the transportation of the operand generated by u to the input operand queue of v, which is the sum of the delay values of the routing switches and wires along the $u \rightarrow v$ path. Then, the total execution time is estimated as follows:

$$\text{tot\_exec} = \max \{\text{exec}(p) | \forall p \in G\}. \quad (1)$$

[4]The total execution time is not to be confused with total runtime in this paper. The total execution time refers to the total number of clock cycles needed to finish executing the given DFG on a target dataflow machine, which is heavily dependent upon the quality of DFG mapping solution. The total runtime refers to the total elapsed CPU time to complete the DFG mapping process, which is heavily dependent upon the efficiency of the mapping algorithms. We report both metrics in Section V.

ILP-based Hierarchical Dataflow Graph Mapping

Input: dataflow graph G and architecture A;
Output: mapping of G to A;

1: cluster nodes in G;
2: place each cluster in G on a cluster in A;
3: route clusters in G using inter-cluster wires in A;
4: **for** (each cluster $c \in G$ placed in cluster C)
5: perform terminal propagation for c;
6: place each node in c on an element in C;
7: route nodes in c using intra-cluster wires in C;

Fig. 3. Overview of our ILP-based hierarchical DFG mapping algorithm that consists of DFG clustering, placement, and routing.

In other words, we compute the weighted longest path delay from a mapped DFG, where the access frequency information is used as the weight on each DFG edge. Our path analysis is based on the standard DFS-based longest path calculation, where the nodes are visited in a topological order for their weighted path lengths computation. This algorithm runs in $O(n)$ due to the depth-first search. Note that this approach also handles cycles in DFG naturally. Since our profiling determines the number of times a given cycle is executed from a given input set, our weighted path length calculation accurately computes the total execution time of each path.

Note that there exist FFs on the intracluster/intercluster communication channels. This means the clock period of the target dataflow machine is fixed. Thus, the goal of our DFG mapping process is to minimize the total number of clock cycles required to evaluate the DFG as described in (1). DFG placement and routing determines which set of routing switches and wires is mapped to each DFG edge. This in turn determines the delay (i.e., total clock cycles) of each edge (i.e., communication) in the DFG. Our approach to minimize the total DFG execution time is by placing the frequently communicating DFG nodes closer together and route them using as few switches/wires as possible. The minimization of pure wirelength, which is the distance between the source and sink node of each edge in a mapped DFG, does not necessarily translate to the total execution time reduction since some DFG edges are not used as often as others during the DFG execution. This is why our "profile-aware" wirelength introduced in Section IV-B. is proven to be more effective.

IV. DFG Mapping Algorithm

A. Overview of the Algorithm

An overview of our hierarchical ILP-based placement and routing algorithm is shown in Fig. 3. Due to the size of the DFGs being considered, it is infeasible to optimally solve the mapping problem using ILP. Therefore, our DFG mapping algorithm is based on a clustering paradigm. At each stage, except clustering, the solution is found using the ILP formulations. The first step in our algorithm is clustering the DFG. Then, the DFG clusters are mapped to architectural clusters. This is followed by intercluster routing. Next, the DFG nodes in each cluster are individually mapped onto architectural elements while taking into consideration terminal propagation information available from the prior cluster-level placement and routing stage. This mitigates much of the nonoptimality introduced during the clustering process.[5] Finally, intracluster routing is done to obtain the final solution. Fig. 4 shows an illustration of our hierarchical DFG mapping process.

[5]A quantitative study on exactly how much of nonoptimality our clustering stage introduces is interesting but not within the scope of this paper.

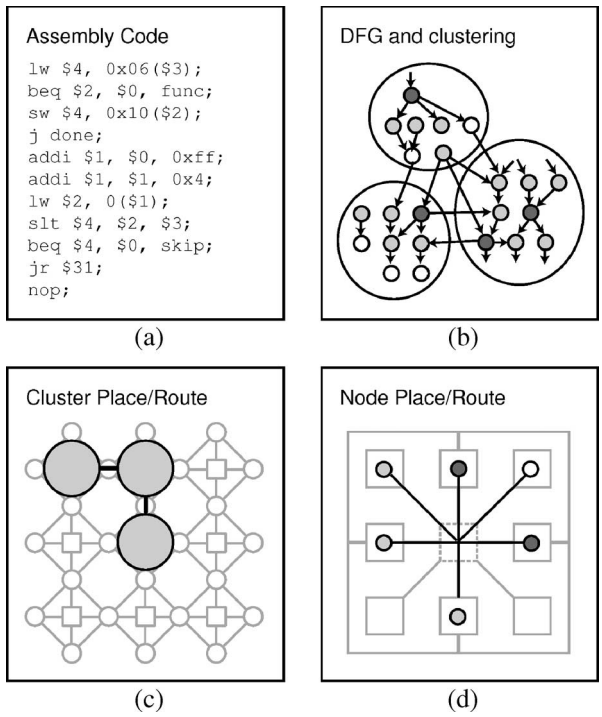

Fig. 4. Illustration of DFG mapping. (a) Sample assembly code. (b) Its DFG and clustering. (c) Placement and routing at cluster level. (d) Placement and routing at element level.

B. Profile-Weight Computation

Our C-code simulation discussed in Section III-D is used to collect profile information. This information allows us to compute profile weight for each edge in the corresponding DFG as follows:

$$\text{p\_wgt}(e) = \left\{ \frac{\text{freq}(e)}{\max\_\text{freq}} \right\}^k \tag{2}$$

where $\text{p\_wgt}(e)$ is the profile weight of edge e, and $\text{freq}(e)$ is the access frequency of $e(x, y)$, i.e., the total number of times the instruction i has provided operands to instruction j during the C-code simulation. $\max\_\text{freq}$ is the maximum among all $\text{freq}(e)$ values. These $\text{p\_wgt}(e)$ values are used during the entire DFG mapping process including clustering, placement, and routing to minimize the total execution time. The weights in the above equation are normalized and then raised to a variable power factor. Related experiments showed that $k = 5$ was an empirically good choice for the power factor. This power factor had the effect of pushing weights that were close to zero even closer to zero. This effects the solution by concentrating computational effort more on the edges that have the highest weight. DFGs characteristically have large dispersion among profiled weight [0–10 000]; thus, the power factor ensures that only edges that have a large impact on execution time are considered. The total simulation time depends on the size of DFG as well as the input sets.

A similar concept used in timing-driven ASIC/FPGA placement is "net weighting" [21], [22]. In this method, static timing analysis is performed to compute timing slack for each circuit element, which is then used to compute weights for each net so that the nets with smaller (or even negative) slacks are given higher weights (i.e., priority). The placement engine then tries to minimize the weighted wirelength so that the delay along the timing critical nets is reduced. The goal of timing-driven net weighing is the same as that of our profile-weighing scheme, where we identify and tackle performance-critical communication paths. However, our profile weight is not based on

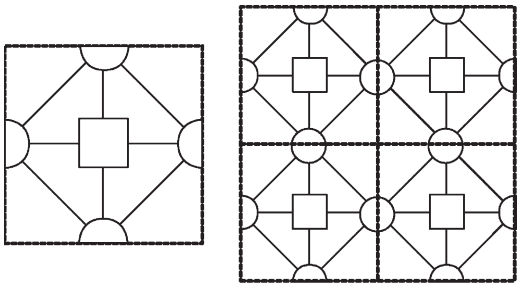

Fig. 5. Illustration of DFG clusters. (a) Single DFG cluster with eight arithmetic, eight memory, and 12 multiplexing DFG nodes packed together. (b) These DFG clusters can be mapped to anywhere on the target dataflow machine, i.e., they are under no type constraint.

timing analysis but on application simulation using program input sets. In addition, timing analysis is typically performed several times to update the changes on the placement, whereas the profile weights are typically computed only once before DFG placement starts. Finally, the profiling result is heavily dependent on the characteristics of input set so that the mapping solution quality may be poor if the input set fails to represent the program behavior correctly.

C. Clustering Algorithms

There exist two kinds of architectural clusters in our dataflow machine: arithmetic and memory. Thus, it is possible to pack eight arithmetic and four multiplexing DFG nodes into an "arithmetic cluster" and four memory and four multiplexing DFG nodes into a "memory cluster." Instead, we pack eight arithmetic, eight memory, and 12 multiplexing DFG nodes into a single cluster, as illustrated in Fig. 5. The reason is that these single type clusters can be mapped to anywhere on the target dataflow machine without any architectural cluster type constraint. This reduces the number of constraints used in ILP formulation, which in turn translates to faster runtime.

Three different clustering algorithms, i.e., random, edge-separability based (ESC) [23], and greedy, were investigated for solution quality. A good clustering solution is one that sufficiently utilizes each architectural cluster while retaining routability and minimizing impact upon the optimality of the placement solution. Type constraints were met during the clustering process to ensure that each cluster was mappable to any architectural cluster. In random clustering, the DFG nodes were randomly assigned to architecture clusters as long as it was feasible to add that node considering space and type constraints. In greedy clustering, the DFG nodes were first ordered by highest profiling frequency on incident edges. The list was then iterated through from largest weight to smallest weight. In each iteration, the current cluster was combined with the node connected to it with the highest profiling frequency until that cluster was full. ESC is an advanced clustering algorithm that clusters the graph using an ordering of the edges based on a metric related to the mincut.

For both ESC and greedy clustering, a post process was done that compared all pairs of clusters to determine whether or not they could be combined considering space constraints. This post process improved the utilization of the architectural clusters while simplifying the top-level cluster placement. Because utilization of the architectural clusters is high when using all the clustering algorithms, this post cluster merging process did not detectably alter wirelength.

D. ILP-Based Placement Algorithm

The basic idea behind the ILP placement formulation is to minimize weighted wirelength using Manhattan distance and a mapping matrix while following the type constraints. The parameters used in the ILP-based placement formulation are defined as follows.

1) V is the set of DFG nodes, and E is the set of directed edges, where edge (i, j) represents an edge from DFG node i to j.
2) N is the set of architectural nodes.
3) $C_{i,j}$ is the profile-weighted Manhattan distance between DFG node i and j, where $P_{i,j}$ and $Q_{i,j}$ are the horizontal and vertical components of $C_{i,j}$, respectively.
4) Map is a mapping matrix, where rows are associated with DFG nodes, columns are associated with architecture nodes, and a 1 in position i, j implies that DFG node i is mapped to architecture node j.
5) $\lambda_{i,j}$ is the statistical traffic on DFG edge (i, j) collected by the compiler.
6) $type(V_i)$ is the type of DFG node i, and type(N_j) is the type of architectural node j. There exist three types of DFG nodes and architectural nodes: arithmetic, memory, and multiplexing.
7) c_j is the capacity of architectural node j. Each arithmetic cluster in the architecture can contain eight arithmetic-type DFG nodes and four multiplexing-type DFG nodes. Each memory cluster in the architecture can contain four memory-type DFG nodes and four multiplexing-type DFG nodes.
8) X_j is the x position of architectural node j, and Y_j is the y position of architectural node j.

Our ILP-based DFG placement is formulated as follows:

$$\text{Minimize} \sum_{(i,j)\in C} C_{ij} \tag{3}$$

subject to

$$\text{map}_{i,j} = \{0, 1\}, \qquad i \in V,\ j \in N \tag{4}$$

$$\sum_{j} \text{map}_{i,j} = 1, \qquad i \in V,\ j \in N \tag{5}$$

$$\sum_{i} \text{map}_{i,j} \le c_j, \qquad i \in V,\ j \in N \tag{6}$$

$$\sum \text{map}_{i,j} \le 0, \qquad \text{type}(V_i) \neq \text{type}(N_j). \tag{7}$$

For given $i, j \in V$ and $k, l \in N$, we compute the profile-weighted wirelength as follows:

$$P_{i,j} = \sum_{k,l} \left[\lambda_{i,j} * (\text{map}_{i,k} * X_k - \text{map}_{j,l} * X_l)\right] \tag{8}$$

$$Q_{i,j} = \sum_{k,l} \left[\lambda_{i,j} * (\text{map}_{i,k} * Y_k - \text{map}_{j,l} * Y_l)\right] \tag{9}$$

$$C_{i,j} = |P_{i,j} - Q_{i,j}|. \tag{10}$$

The first constraint forces the integrality of the mapping matrix. Constraint (5) guarantees that each DFG node is mapped to exactly one architectural node. Constraint (6) guarantees that each architectural node has at most its capacity mapped to it. The matrix P corresponds to the weighted X distance between two modules, whereas the matrix Q corresponds to the weighted Y distance between two modules. Finally, (7) ensures that type constraints are observed when finding a solution.

There exist two differences between cluster-level and element-level ILP placements. First, the type constraint (7) is ignored during cluster-level placement as explained in Section IV-C, whereas element-level placement strictly enforces this constraint. Second, terminal propagation detailed in Section IV-F is only used during element-level ILP placement. In case the $\lambda_{i,j}$ term is omitted from (8) and (9), the previous ILP formulation becomes pure wirelength-driven placement, which is used in our comparative study presented in Section V.

E. ILP-Based Routing Algorithm

The parameters used in the ILP-based routing formulation are defined as follows.

1) V is the set of DFG nodes, and E is the set of directed edges, where edge (i, j) represents an edge from DFG node i to j.
2) N is the set of architectural nodes.
3) Let flow be an electronic signal flow sending from source to sink, where rows and columns are associated with architecture nodes based on each path. $f_{i,j,k}$ is set to 1 if there is a flow from architecture node i to architecture node j on DFG edge k, and 0 otherwise.
4) λ_k is the statistical traffic on DFG edge k.
5) $c_{i,j}$ is the capacity of architectural channel i, j. Note that $c_{i,j}$ is equal to $c_{j,i}$.
6) $b_{i,k}$ is the summation of all flows into and out of architectural node i for DFG edge k.

Our ILP-based DFG routing is formulated as follows:

$$\text{Minimize} \sum_{(i,j,k)} \lambda_k \cdot f_{i,j,k} \tag{11}$$

subject to

$$\sum_{i,j,k} f_{i,j,k} + f_{j,i,k} \leq c_{i,j}, \qquad i,j \in A,\ k \in E \tag{12}$$

$$\sum_{l} f_{i,l,k} - \sum_{j} f_{j,i,k} = b_{i,k} \leq \sum_{m} c_{i,m}, \qquad i \in N,\ j \in A \tag{13}$$

$$f_{i,j,k} \geq 0 \tag{14}$$

$$\lambda_{i,j} \geq 0. \tag{15}$$

This ILP-based routing formulation is based on multicommodity flow formulation. The objective is to minimize the number of hops required when routing from a source node to a sink node while maintaining the channel capacity constraint. From the formulation, constraint (12) guarantees that the inflow and outflow of each wire channel do not exceed the capacity limit. Constraint (13) guarantees that the inflows are equal to the outflows for all nodes in the route and equal to 1 for the supply node and -1 for the demand node. Only the supply node can have the surplus outflow, and only the demand node can have the surplus inflow. The inflow is equal to the outflow for the intermediate nodes. The profile-weighted wirelength is the minimization objective. The previous ILP formulation is used for both cluster-level and element-level routings. In case the $\lambda_{i,j}$ term is omitted from (11), the previous ILP formulation becomes pure wirelength-driven routing, which is used in our comparative study presented in Section V.

F. Terminal Propagation

During the element-level placement, the DFG nodes in each DFG cluster are placed on arithmetic or memory element based on its type. Some of the DFG nodes to be placed may have connections to other DFG nodes outside the current DFG cluster under consideration. In this case, the placement of these external neighboring nodes can be exploited to reduce interconnect length. More specifically, if there is a DFG node n that has many connections to other DFG nodes placed to the left, placing n closer to the left boundary of the target architectural cluster instead of other location will reduce the overall wirelength. This concept, so-called "terminal propagation" [24], is frequently used in partitioning-driven placement approach for VLSI circuits.

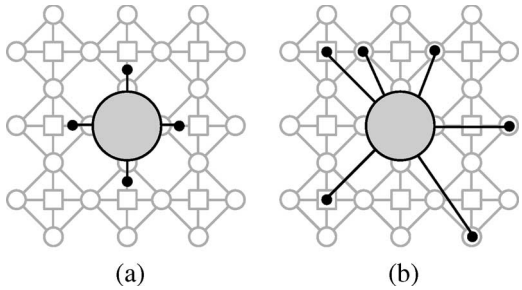

Fig. 6. Illustration of terminal propagation algorithm. (a) Cardinal style. (b) Dispersed style.

Two styles of terminal propagation are studied: 1) cardinal and 2) dispersed. Fig. 6 illustrates these two styles. Cardinal style places four terminals on the top, bottom, left, and right sides of the target architectural cluster C. If an external neighbor j of a DFG node $i \in C$ is located above C, a single top terminal T_t is used to represent j. T_t then pulls i closer to the top boundary of C. In dispersed cycle style, the actual location of the external neighbors is used as the terminal location. After all terminal locations are defined, the location of these terminals is added to the ILP formulation as additional constraints. In case of cardinal style, we add the following constraints for a given DFG node i in architectural cluster C, i.e.,

$$\text{map}_{j,k} = 1,\ \forall j \notin C \text{ and } e(i,j) \in E,\ k \in \{T_t, T_b, T_l, T_r\}$$

where T_t, T_b, T_l, and T_r denote the top, bottom, left, and right terminals for C. These terminals are located at the center of the architectural clusters directly above, below, to the left, and to the right of C. Note that the above equations are added for each DFG node in C. In case of dispersed style, we add the following constraints for a given DFG node i in architectural cluster C:

$$\text{map}_{j,k} = 1,\ \forall j \notin C \text{ and } e(i,j) \in E, \quad k \in V$$

where k is the arithmetic cluster j placed into (during cluster-level placement), and j is the external neighbor of i. After the related constraints are added, the placement is carried out while trying to minimize the weighted wirelength connecting not only the internal DFG nodes but also the external neighbors.

G. Example

This section provides an example of our C-to-mapping flow. We are trying to map the following C-code:

```
for (i = 0; i < 10; i++) {
  if (i < 5)
    sum = sum + 1;
  else
    sum = sum + 2;
}.
```

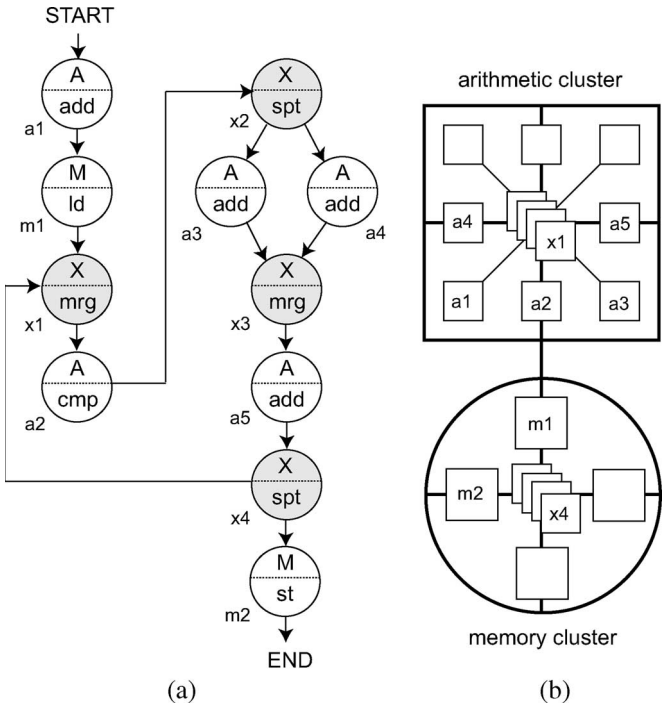

Fig. 7. (a) DFG of our sample C-code, where the gray nodes represent multiplexing nodes. (b) Its mapping, where nodes x_2 and x_3 (hidden under x_1 and x_4) are mapped to the arithmetic cluster.

TABLE I
APPLICATION CHARACTERISTICS

graph	function	#ao	#nodes	#edges	#aclus
gzip	fill_windows	140	217	239	5x5
vpr	get_non_upd	228	382	429	8x8
mcf	price_out	269	435	548	8x8
equake	phi0	37	54	54	5x5
parser	region_valid	504	755	892	8x8
vortex	mem_getword	79	126	128	5x5
bzip2	spec_putc	56	81	85	8x8
twolf	ucxx1	353	574	590	8x8

The above C-code is compiled to the following assembly program:

```
        r0 < −0;
        r1 < −i;
        r2 < −sum;
        add r1, r0, r0;
        ld r2, sum;
loop :  bge r1, 5, br1;
        add r2, r2, 1;
        jmp br2;
br1 :   add r2, r2, 2;
br2 :   add r1, r1, 1;
        blt r1, 10, loop;
        st r2, sum.
```

Fig. 7 shows the corresponding DFG and its mapping.

V. EXPERIMENTAL RESULTS

The framework was run on Pentium IV 2.4-GHz dual processor systems. It was written using a combination of C++ compiled with g++ version 3.2.2 and perl scripts using perl version 5.8.0. Table I illustrates the characteristics of our SPEC2000 benchmark applications. We report: 1) the name of the function that is selected; 2) the number of arithmetic nodes (#ao), total nodes (#nodes), and edges (#edges) in each DFG; and 3) the dimension of the target dataflow machines used in terms of the number of arithmetic clusters (#aclusters). Our DFGs are derived based on the functions that most capture the benchmark behavior reported in [25].

Throughout the experiment, wirelength was measured in terms of Manhattan distance as discussed in Section III-C. The total execution time was measured in terms of million clock cycles (the clock frequency is fixed throughout our experiment). The total runtime was measured in seconds. Since the target dataflow machine is fixed, our wirelength results are not meant to imply routing cost. Instead, our wirelength is used in our placement to improve the overall execution time of DFG. In addition, a shorter wirelength translates to lower usage of routing resource (wires and switches), which in turn translates to lower communication delay and power consumption.

The solutions to the ILPs were found using the GNU Linear Programming Kit [26] version 4.5 glpsol executable. While finding the optimal solution, the linear program solver first finds a linear optimal solution and then attempts to make the solution integral. Because the linear solution can be equal to zero in some cases, the integer optimization step could iterate through every possible solution before giving up, which would cause the runtime to be extremely long. Therefore, it was necessary to limit the runtime to 1 h per ILP for standard and 2 h per ILP for larger cases. Typically, the solver iterates around the same minimum value for millions of iterations before being terminated; thus, we believe the nonoptimality introduced by this time limit is negligible.

ILP problems can be solved near optimally. Thus, the largest factor introducing nonoptimality in the solution algorithm is the clustering process. A comparison of the three clustering algorithms is given in Table II. It can be seen that the random clustering algorithm produces solutions that are significantly worse in execution time than that of the best result. Wirelength is also significantly worse and is in many cases unroutable because of this. This is due to the fact that the random clustering algorithm frequently places nodes connected with edges in different clusters, whereas ESC and the greedy clustering algorithm specifically target nodes connected by edges for clustering. When comparing ESC versus greedy clustering, it can be seen that greedy clustering produces slightly worse solutions in terms of execution time. This is explained when one analyzes the wirelength numbers and sees that ESC generally has better wirelength. We use ESC clustering for all subsequent experiments.

When comparing the number of clusters, it is very noticeable that the three different clustering algorithms produce exactly equal numbers of clusters. When comparing greedy and random clustering, this is very probable because both algorithms simply pack nodes into clusters until more nodes cannot be fit. When delving further into the subject, it is revealed that DFGs are forests and that the number of nodes in each tree of the forest could be as small as one. These nodes with no edges correspond to control instructions or no-ops inserted by the compiler that do not have data dependencies with the rest of the instructions. Because of the forest-like nature of DFGs, both ESC and greedy clustering will arrive at similar numbers of clusters if a large percentage of trees in the forest is of sufficiently small size so as to fit into a single architectural block. This will be only compounded by the post process. However, our analysis shows that although the number of clusters is the same, the size and nodes of each cluster are different between all the algorithms.

A comparison of ILP-based profile-driven versus wirelength-driven placement (Table III) and routing (Table IV) is given in their respective tables. Our baseline algorithm is wirelength-driven algorithms, where the objective functions do not utilize the profile weights. Our profile-driven placement algorithm outperforms the traditional

TABLE II
IMPACT OF CLUSTERING. WE REPORT TOTAL NUMBER OF CLUSTERS FORMED (#CLUST), WIRELENGTH (WIRE), EXECUTION TIME (EXEC), AND TOTAL ELAPSED CPU TIME (IN SECONDS) ALL AFTER ROUTING. WE USE OUR PROFILE-DRIVEN ILP PLACER

	random clustering				ESC clustering				greedy clustering			
bench	#clust	wire	exec	CPU	#clust	wire	exec	CPU	#clust	wire	exec	CPU
gzip	18	1984	24690	3663	18	263	7550	3636	18	425	5509	3654
vpr	29	unroutable			29	1135	8689	3881	29	1197	9887	3749
mcf	34	unroutable			34	2170	31764	36040	34	2170	44470	36942
equake	5	206	9612	172	5	48	3162	97	5	91	3180	71
parser	63	unroutable			63	3206	16396	4650	63	3930	21234	36060
vortex	10	711	24304	3623	10	181	6867	3647	10	366	6018	3623
bzip2	7	400	7080	3612	7	239	2202	158	7	207	2642	141
twolf	44	unroutable			44	1331	593	4639	44	1578	662	1499

TABLE III
ILP PLACEMENT RESULTS BASED ON PURE WIRELENGTH OBJECTIVE VERSUS PROFILE-WEIGHT OBJECTIVE. WE REPORT THE TOTAL EXECUTION TIME WITHOUT INTERCONNECT LATENCY UNDER "NO WIRE" COLUMN

	wirelength-driven			profile-driven			no wire
bench	wire	exec	CPU	wire	exec	CPU	exec
gzip	114	5158	2815	168	5536	2472	3021
vpr	489	6729	4327	625	6250	2627	3197
mcf	772	37668	4633	2170	22872	25828	12631
equake	24	3162	150	30	3162	64	2464
parser	1577	14840	3945	1678	9869	3394	6968
vortex	63	6009	2352	114	6009	2285	4292
bzip2	41	2789	412	129	2202	125	1762
twolf	572	512	3656	747	406	3239	252
RATIO	1.00	1.00	1.00	1.77	0.86	1.31	0.56

TABLE IV
ILP ROUTING RESULTS BASED ON PURE WIRELENGTH OBJECTIVE VERSUS PROFILE-WEIGHT OBJECTIVE. OUR LP-BASED ROUTING RESULT SERVES AS A LOWER BOUND

	wirelength-driven			profile-driven		
bench	wire	exec	CPU	wire	exec	CPU
gzip	182	6793	4002	263	7550	3636
vpr	867	9968	6658	1135	8689	3881
mcf	1439	63582	6400	2170	31764	36040
equake	37	3162	221	48	3162	97
parser	3034	26999	5241	3206	16396	4650
vortex	107	7726	3829	181	6867	3647
bzip2	63	3377	610	239	2202	158
twolf	927	857	5153	1331	593	4639
RATIO	1.00	1.00	1.00	1.69	0.79	1.32

wirelength-only-driven algorithm by 14% in average execution time calculated before routing was done. Our profile-driven router outperforms the traditional router by 21% in average execution time calculated after routing was done. The wirelength is almost tripled in benchmark "mcf" in Table III. However, the total execution time is reduced by 40%. This is a strong evidence that the profiling information available from DFG simulation, not pure wirelength, is more effective in reducing the overall execution time of DFGs. The benchmark "bzip2" is a similar example.

In most benchmarks, CPU time of profile-driven ILP-based placement is less than that of wirelength-driven ILP-based placement. Some paths in the given DFG are rarely executed compared to other frequently executed paths. The "for" and "while" loops are a good example of frequently executed paths. This causes the profile weight on rarely executed paths become very small or sometimes zero after the normalization discussed in (2). The presence of zero weight edges typically improves the runtime of ILP-based methods since these edges are ignored during the computation. Upon inspection, one will notice that "gzip" has longer execution time for the profiling case. This occurs because our profiling method relies on capturing real data behavior. If the data change drastically from the training input set, then profiling may have negative impact on performance.

TABLE V
SA [27] VERSUS ILP-BASED DFG PLACEMENT

	SA+wire		SA+profile		ILP+profile	
bench	wire	exec	wire	exec	wire	exec
gzip	380	6950	433	4940	168	5536
vpr	868	8530	1006	6550	625	6250
mcf	1125	39430	1409	33420	2170	22872
equake	42	3190	59	3160	30	3162
parser	2590	10580	2827	8540	1678	9869
vortex	140	7440	165	6010	114	6009
bzip2	69	2500	127	2200	129	2202
twolf	1691	600	1917	480	747	406
RATIO	1.00	1.00	1.27	0.83	0.95	0.79
TIME	770		1144		5004	

Table III provides the total execution time when the interconnect latency is completely ignored, i.e., we consider the computation delay only. Since the interconnect latency is determined by the placement, this result serves as a lower bound on the placement quality. We note from Table III that our profile-driven placer obtains results that are within 30% of this lower bound.

Table V shows a comparison between our ILP-based DFG placement and SA-based DFG placement [27]. In SA-based placement, we use either the pure wirelength or profile-weighted wirelength as our objective.[6] We draw two major conclusions from Table V. First, the profile-driven methods (SA + profile and ILP + profile) improve the total DFG execution time results of nonprofile method (SA + wire) by 17% and 21% on average. This shows that it is critical to utilize the profile information to improve the total DFG execution time. However, wirelength has increased by 27% in SA + profile. We note that the range of profile-based edge weights is quite huge: [0, 10 000]. This in turn causes a huge wirelength penalty in our profile-driven placement. Second, ILP + profile improves the wirelength results of SA + profile by 32% on average. A shorter wirelength translates to lower usage of routing resource (wires and switches), which in turn translates to lower communication delay and power consumption. Thus, the combination of both profile information and ILP approach is proven to be effective in both wirelength and execution time optimization. Note that the ILP-based results are suboptimal for some benchmarks due to the nonoptimality introduced by the clustering stage. This explains why ILP-based approach generates worse results than SA-based approach for some benchmark.

Table VI compares the two styles of terminal propagation: cardinal style and dispersed style. The table shows that both techniques provide a comparable result. However, cardinal style requires only four additional pseudo nodes and easier to implement.

[6]To the best of our knowledge, the work presented in [27] is the only existing work that allows a meaningful comparison.

TABLE VI
IMPACT OF TERMINAL PROPAGATION. WE USE OUR PROFILE-DRIVEN ILP PLACER

	cardinal		dispersed	
bench	wire	exec	wire	exec
gzip	263	7550	261	7322
vpr	1135	8689	1125	8689
mcf	2170	31578	2173	31764
equake	48	3162	48	3162
parser	3206	16396	3206	16396
vortex	181	6867	180	6867
bzip2	239	2202	238	2202
twolf	1331	593	1331	593

VI. CONCLUSION

Configurable dataflow architectures recently became more popular due to their ability to extract more parallelism and reduce the impact of wire delay. In this paper, we proposed ILP-based placement and routing algorithms that use clustering to reduce problem complexity while still retaining solution quality. The optimization process is guided by profiling information available from the compiler. Our goal is to minimize the total execution time of the given application represented by a DFG under architectural constraints. Our profile-driven ILP algorithm reduces the total execution time of benchmark applications compared to the conventional wirelength-driven ILP approach and outperforms SA-based approach.

Our proposed solution is not without limitation. First, the half-perimeter bounding box (HPBB) metric used in our placement is not accurate due to the diagonal intercluster routes and nontrivial intracluster routes. The rationale behind our choice of HPBB is that the shorter distance between the source and sink nodes tends to require less routing resources and thus reduce the delay. However, the actual delay depends on the detailed routing result. Thus, an integrated placement and routing approach will provide more accurate routing resource usage to guide placement but at the cost of longer runtime. One problem with ILP-based approach is that it does not scale well with the size of the problem. One possible solution is to relax the integer constraints, solve LP, and apply a rounding heuristic to obtain integer results. We can optionally iterate the LP + rounding until we obtain satisfactory results.

REFERENCES

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[2] S. Thakur, Y.-W. Chang, D. Wong, and S. Muthukrishnan, "Algorithms for an FPGA switch module routing problem with application to global routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 16, no. 1, pp. 32–46, Jan. 1997.

[3] L. Behjat and S. Chiang, "Fast integer linear programming based models for VLSI global routing," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2005, pp. 6238–6243.

[4] Z. Li, W. Wu, and X. Hong, "Congestion driven incremental placement algorithm for standard cell layout," in *Proc. Asia and South Pacific Des. Autom. Conf.*, 2003, pp. 723–728.

[5] M. Narasimhan and J. Ramanujam, "Improving the computational performance of ILP-based problems," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 1998, pp. 593–596.

[6] X. Yang, R. Kastner, and M. Sarrafzadeh, "Congestion reduction during placement based on integer programming," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 2001, pp. 573–576.

[7] J. B. Dennis, "A preliminary architecture for a basic dataflow processor," in *Proc. IEEE Int. Symp. Comput. Architecture*, 1975, pp. 126–132.

[8] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor," in *Proc. IEEE Int. Symp. Comput. Architecture*, 1989, pp. 46–53.

[9] V. Grafe, G. Davidson, J. Hoch, and V. Holmes, "The epsilon dataflow processor," in *Proc. IEEE Int. Symp. Comput. Architecture*, 1989, pp. 36–45.

[10] G. Papadopoulos and D. Culler, "Monsoon: An explicit token-store architecture," in *Proc. IEEE Int. Symp. Comput. Architecture*, 1990, pp. 82–91.

[11] K. Sankaralingam *et al.*, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proc. IEEE Int. Symp. Comput. Architecture*, 2003, pp. 422–433.

[12] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proc. IEEE Micro*, 2003, pp. 291–302.

[13] J. J. Granacki and M. Vahey, "MONARCH: A high performance embedded processor architecture with two native computing modes," in *Proc. High Performance Embedded Comput.*, 2002.

[14] M. Taylor *et al.*, "Evaluation of the raw microprocessor: An exposed wire delay architecture for ILP and streams," in *Proc. IEEE Int. Symp. Comput. Architecture*, 2004, pp. 2–13.

[15] J. Cardoso, "On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures," *IEEE Trans. Comput.*, vol. 52, no. 10, pp. 1362–1375, Oct. 2003.

[16] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 579–590, Jun. 1999.

[17] R. Nagarajan, S. Kushwaha, D. Burger, K. McKinley, C. Stephen, and W. Keckler, "Static placement, dynamic issue (SPDI) scheduling for EDGE architectures," in *Proc. Int. Conf. Parallel Architecture and Compilation Tech.*, 2004, pp. 74–84.

[18] Trimaran. [Online]. Available: http://www.trimaran.org

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. IEEE Micro*, 1992, pp. 45–54.

[20] H. Khalid, "Validating trace-driven microarchitectural simulations," *IEEE Micro*, vol. 20, no. 6, pp. 76–82, Nov./Dec. 2000.

[21] T. Kong, "A novel net weighting algorithm for timing-driven placement," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 2002, pp. 172–176.

[22] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2000, pp. 203–213.

[23] J. Cong and S. K. Lim, "Edge separability based circuit clustering with application to multi-level circuit partitioning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 3, pp. 346–357, Mar. 2004.

[24] A. Dunlop and B. Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. CAD-4, no. 1, pp. 92–98, Jan. 1985.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002, pp. 45–57.

[26] GLPK, *GLPK (GNU Linear Programming) Kit*. [Online]. Available: http://www.gnu.org/software/glpk/glpk.html

[27] M. Ekpanyapong, M. Healy, and K. Lim, "Placement for configurable dataflow architecture," in *Proc. Asia and South Pacific Des. Autom. Conf.*, 2005, pp. 1127–1130.