MILP-based Placement and Routing for Dataflow Architecture

Michael Healy, Mongkol Ekpanyapong, and Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology
{mbhealy, pop, limsk}@ece.gatech.edu

***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, we propose MILP-based placement and routing algorithms for mapping dataflow graphs to dataflow machines. 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 dataflow graph under architectural constraints. We propose a hierarchical method to handle the complexity of the initial MILP formulation. Our profile-driven MILP algorithm reduces the total execution time of benchmark applications compared to the conventional wirelength-driven approach on average by 18%.**

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 ALUs, storage units and communication paths over a 2-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.

Dataflow architectures have the potential to achieve very high performance when parallelism is exploited at the programming level. While this is not possible using standard benchmarks the compiler is capable of extracting suitable parallelism for dataflow architectures. There has been ongoing research that targets non-streaming applications such as TRIPS [1] and WaveScalar [2]. Dataflow architectures that target streaming appliactions include TRIPS and MONARCH [3].

An efficient mapping of applications to the elements of a dataflow architecture would place frequently and time-critically communicating parts of the computation close to each other, thereby delivering very high performance. However, 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 contributions of this paper are MILP-based placement and routing algorithms for mapping dataflow graphs to dataflow machines. 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 dataflow graph under architectural constraints. We propose a hierarchical method to handle the complexity of the initial MILP formulation. Our profile-driven MILP algorithm reduces the total execution time of benchmark applications compared to the conventional wirelength-driven approach on average by 18%. Related works are hard to find. A recent work [4] performs dataflow graph clustering targeting distributed register-file machine and provides a survey on related works in compiler community.

The rest of this paper is organized as follows. Section II presents the basics of data flow graphs and architectures. Section III presents the problem formulation . Our dataflow graph mapping algorithm is described in Section IV. Experimental results are presented in Section V, and we conclude in Section VI.

II. Preliminaries

A. Dataflow Architecture and Graph

A MONARCH-like architecure, with some WaveScalar extensions is targeted in this paper. Figure 1(a) shows an illustration of the generic dataflow architecture that is used. Squares in the dataflow fabric represent arithmetic clusters and circles represent memory clusters. Figure 1(b) shows the extension made to the ALUs, which is similar to the ALUs used in WaveScalar. In each ALU, the input operands contain a tag field which 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

0-7803-9362-7/05/$20.00 ©2005 IEEE

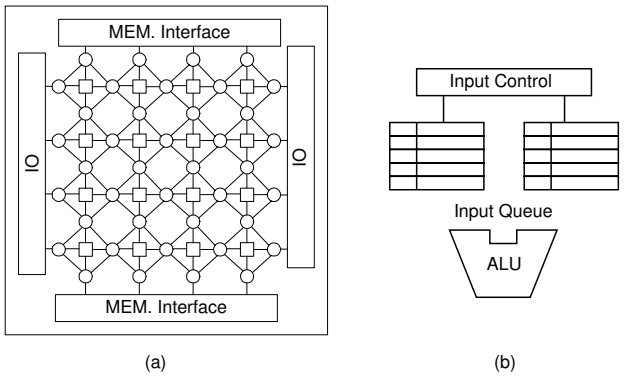

Fig. 1. (a) Example of generic dataflow fabric, where squares epresent arithmetic clusters and circles represent memory clusters, (b) extended processing element used in our dataflow architecture.

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.

Dataflow graphs (DFG) are used to identify which instructions produce data needed by other instructions. If-conversion [5] 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 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.

III. Problem Formulation

A. Design Flow

First, an application is fed into the system by the front-end compiler. The dataflow graph 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 [6] such that the placer reads the annotated dataflow graph, 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, placer, 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.

B. Dataflow Graph Mapping Problem

Our dataflow graph 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: arithmetic, memory, and multiplexing nodes. Each node is associated with a non-uniform delay, where more complex operations such as multiplcation 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.

Our dataflow architecture contains two-levels of hierarchy: element-level and cluster-level. Each element can accommodate (and thus execute) a single dataflow node, and each cluster contains multiple elements. We assume all elements in the same cluster are located at the same position. There are three types of architectural clusters: arithmetic, memory, and multiplexing clusters. We model the dataflow architecture using a graph $A(N, W)$, where N is the set of clusters and W is the set of wires connecting the clusters. In addition, each cluster $c \in N$ is further represented via $A_c(N_c, W_c)$, where N_c is the set of elements contained in c, and W_c is the set of wires connecting these elements. Each cluster is given a unique 2D location and is under a capacity constraint. Each inter-cluster wire $w \in W$ and intra-cluster wire $w_c \in W_c$ is under capacity constraint. Formal problem definitions are given as follows:

Definition 1 (Dataflow Graph Placement): we map each DFG node $v \in V$ to a unique architecture node $n \in N$ such that the type and capacity constraints are satisfied.

Definition 2 (Dataflow Graph Routing): we map each DFG edge $e \in E$ to a set of wires from W (inter-cluster) and/or W_i (intra-cluster) such that the capacity constraint is satisfied.

Our minimization objective includes wirelength and profiling weight. Wirelength is calculated as the half perimeter of the bounding box among each edge in G (Manhattan Distance). The profiling weight is equal to the normalized access frequency gathered by the front-end compiler. Therefore the edges of the graphs are both weighted and directed. The calculation of the total execution time is explained in the following section.

C. Execution Time Estimation

Since our dataflow computer has no speculation, we estimate the total execution time of a given application as follows. The estimation is based on the number of times each node and edge are executed. For each path p, the execution time of p,

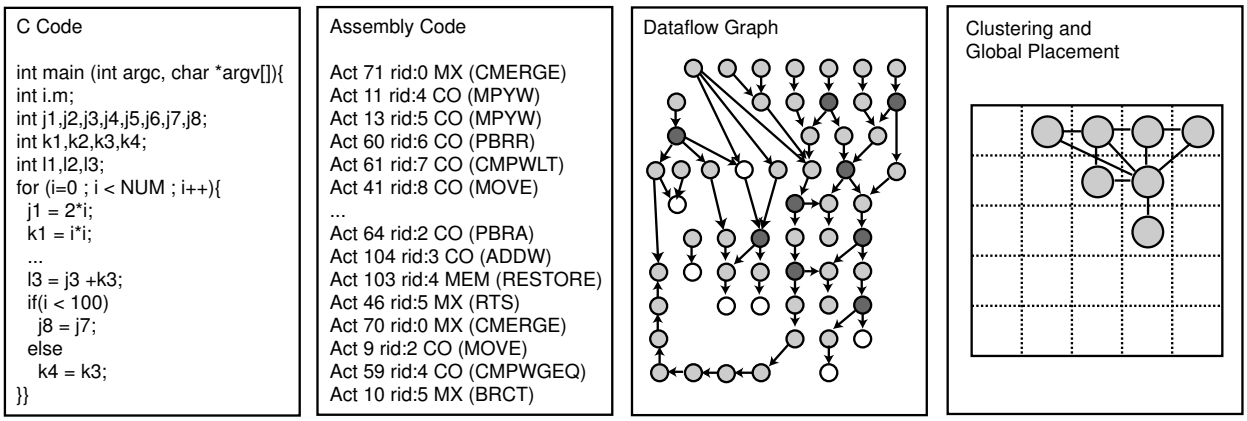

Fig. 2. Illustration of DFG mapping. A sample C code and its assembly code are shown. We also show the corresponding dataflow graph. We cluster the nodes in the DFG and map it onto our 5×5 architecture block set.

denoted $exec(p)$, is computed as follows:

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

where $u, v \in V$ and for all $e \in p$. $freq(e)$ denotes the access frequency collected by profiling.

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. Then the total execution time is estimated as $\max\{exec(p)|\forall p \in G\}$. In other words, we compute the weighted longest path delay where frequency information is used as the weight on each edge. Thus, a topological ordering based $O(n)$ timing analysis is enough to compute the total execution time. To compute the longest path, cycles have to be removed first. However the back edges are included during execution time computation– we add the delay of source gate and the delay of the back edge to all paths that contain this back edge.

To evaluate our placement result, we assume perfect memory and large enough input queue buffers. Note that Wavescalar [2] also assumes perfect L1 data cache and unbounded input queues. Assuming that dataflow architectures are governed by parallelism, performance can be calculated based on how many times each path is executed as well as its delay. Note that TRACE simulation is also similar to this approach. We use a training input set for profiling and a reference input set for performance evaluation.

IV. Dataflow Graph Mapping Algorithm

A. Overview of the Algorithm

Due to the size of the dataflow graphs being considered, it is infeasible to optimally solve the mapping problem using MILP. Therefore, our DFG mapping algorithm is based on a clustering paradigm. At each stage, except clustering, the solution is found using the MILP formulations enumerated below. An architectural cluster consists of a central set of 8 arithmetic nodes and 4 switch nodes and four peripheral groups consisting of 2 switch nodes and 2 memory nodes. Therefore, there are 8 arithmetic, 8 memory, and 12 switch nodes in each cluster. For the sake of clarity, architectural clusters will be referred to as architectural blocks and DFG clusters will be referred to as clusters.

The first step in our algorithm is clustering the DFG. Then the DFG clusters are mapped to architectural blocks. This is followed by high level inter-cluster routing. Next each cluster is individually mapped onto its architectural block while taking into consideration terminal propagation information, explained below, passed to it from the previous routing stage. This mitigates much of the non-optimality introduced during the clustering process. Finally, intra-cluster routing is done to obtain the final solution.

During every step profiling information is used to determine edge weights. The weights are normalized and then raised to a variable power factor. Experimentation showed that 5 was an empirically good choice for the power factor. This power factor had the effect of pushing weights that were closer 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] so the power factor ensures that only edges that have a large impact on execution time are considered.

We experimented with two types of terminal propagation, cardinal direction style and dispersed style. In cardinal direction propagation only 4 extra terminals were added, up, down, left, and right, while in dispersed style terminals were added in the proper direction for every node that had an edge to an extra-cluster node. We found that the difference between the two styles was negligible and so do not report those results here. Figure 2 shows an illustration of the entire flow of our algorithm.

MILP-based DFG Placement

$$\text{Minimize} \quad \sum_{(i,j)\in C} C_{ij} \tag{1}$$

Subject to:

$$map_{i,j} = \{0,1\},\ i \in N, j \in A \tag{2}$$

$$\sum_{j} map_{i,j} = 1,\ i \in N, j \in A \tag{3}$$

$$\sum_{i} map_{i,j} \leq c_j,\ i \in N, j \in A \tag{4}$$

$$\sum map_{i,j} \leq 0,\ type(M_i) \neq type(A_j) \tag{5}$$

$$P_{i,j} = \sum_{k,l} [\lambda_{i,j} * (map_{i,k} * X_k - map_{j,l} * X_l)],\ i,j \in N, k,l \in A \tag{6}$$

$$Q_{i,j} = \sum_{k,l} [\lambda_{i,j} * (map_{i,k} * Y_k - map_{j,l} * Y_l)],\ i,j \in N, k,l \in A \tag{7}$$

$$C_{i,j} \geq P_{i,j} + Q_{i,j},\ i,j \in N \tag{8}$$

$$C_{i,j} \geq P_{i,j} - Q_{i,j},\ i,j \in N \tag{9}$$

$$C_{i,j} \geq -P_{i,j} + Q_{i,j},\ i,j \in N \tag{10}$$

$$C_{i,j} \geq -P_{i,j} - Q_{i,j},\ i,j \in N \tag{11}$$

Fig. 3. MILP-based PCA Placement

B. Clustering Algorithms

Three different clustering algorithms, random, edge-separability based (ESC) [7], and greedy, were investigated for solution quality. A good clustering solution is one that sufficiently utilizes each architectural block while retaining routability and minimizing impact upon the optimality of the placement solution. Type constraints were met during the clustering process in order to ensure that each cluster was mappable to its architectural block.

In random clustering the DFG nodes were randomly assigned to architecture blocks 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 efficient graph-search-based bottom-up clustering algorithm. Unlike existing algorithms that are based on local connectivity information of the netlist such as edge weights, ESC exploits more global connectivity information called *edge separability* to guide the clustering process. For given edge $e = (x, y)$ in an edge-weighted undirected graph $U(V, E_U, W_U)$, edge separability of e is defined as the minimum cutsize among the cuts separating x and y in U. Thus, computing the edge separability for a given edge $e = (x, y)$ is equivalent to finding the x-y mincut. Direct computation of edge separability for all edges in U requires max-flow computation for $|E_U|$ times, which is extremely time-consuming. ESC provides an efficient way to compute a tight lower bound of separability of *all* edges in U in $O(|V| \log |V|)$ time without using any flow computation.

ESC clusters grow from the contraction of *contractible edges*, i.e., the set of edges whose contraction preserves the mincut in U. This graph search algorithm is based on traversing vertices of U according to the *Maximum Adjacency (MA) ordering* of vertices in U. The intuition behind MA ordering is that it chooses a vertex that is *most tightly connected* to the vertices that are already in the order. Then, our search algorithm traverses through vertices in MA ordering while labelling each edge with $q(e)$, a tight lower bound of edge separability. Finally, the contractible edges are computed by comparing $q(e)$ to $\bar{\lambda}(G)$, where $\bar{\lambda}$ denotes the minimum cutsize discovered so far.

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 blocks while simplifying the top level cluster placement. Because utilization of the architectural blocks is high when using all the clustering algorithms this post process did not detectably alter wirelength.

C. MILP-based Placement Algorithm

The basic idea behind the MILP placement formulation is to minimize weighted wirelength using Manhattan distance and a mapping matrix while following the type constraints. The parameters used in the placement MILP formulation shown in Figure 3 are defined as follows. Let N denote the set of DFG

nodes, E denote the set of directed edges where edge (i, j) represents an edge from DFG node i to j, and A denote the set of architectural nodes. Let map be 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. Also, let $\lambda_{i,j}$ be the statistical traffic on DFG edge (i, j), $type(M_i)$ be the type of DFG node i, $type(A_j)$ be the type of architectural node j, and c_j be the capacity of architectural node j. Finally, let X_j be the x position of architectural node j, and Y_j be the y position of architectural node j.

The first constraint forces the integrality of the mapping matrix. Constraint (3) guarantees that each DFG node is mapped to exactly one architectural node. (4) guarantees that each architectural node has at most its' capacity mapped to it. Constraints (6)-(7) compute the standard weighted distance function. The matrix P corresponds to the weighted X distance between two modules while the matrix Q corresponds to the weighted Y distance between two modules. Constraints (8)-(11) linearize the absolute value function for the sum of the X and Y differences. Finally, (5) ensures that type constraints are observed when finding a solution. Additionally, for the purposes of terminal propagation, a psuedo node set can be created and statically mapped to their correct position on the architecture to effect the placement solution as desired above. This is done by simply constraining a particular entry in the matrix corresponding to the psuedo node to have a value of 1.

D. MILP-based Routing Algorithm

The parameters used in the placement MILP formulation defined in Figure (4) are defined as follows. Let N denote the set of DFG nodes, E denote the set of directed edges where edge (i, j) represents an edge from DFG node i to j, and A denote the set of architectural nodes. Let $flow$ be a electronic signal flow sending from source to sink, where rows and columns are associated with architecture nodes based on each path, and a 1 in position i, j, k implies that architecture node i is sending signal to architecture node j on DFG edge k. In addition, let λ_k be the statistical traffic on edge (k). Finally, let c_j be the capacity of architectural channel j, $f_{i,j,k}$ represent the flow from architecture node i to node j from MDFG edge k and $b_{i,k}$ represent the summation of all flows into and out of architectural node i for DFG edge k.

Constraint (13) guarantees that the inflow and outflow of each wire channel do not exceed the capacity limit. (14) guarantees that the inflows are equal to the outflows for all nodes in the route and equal 1 for the supply node and -1 for the demand node. Weighted wirelength is the minimization objective.

MILP-based DFG Routing

$$\text{Minimize} \quad \sum_{(i,j,k)} \lambda_k \times f_{i,j,k} \tag{12}$$

Subject to:

$$\sum_{i,j,k} f_{i,j,k} + f_{j,i,k} \leq c_{i,j} + c_{j,i},\ i, j \in A, k \in E \tag{13}$$

$$\sum_{l} f_{i,l,k} - \sum_{j} f_{j,i,k} = b_{i,k} \leq c_j,\ i \in N, j \in A \tag{14}$$

$$f_{i,j,k} \geq 0 \tag{15}$$

$$\lambda_{i,j} \geq 0 \tag{16}$$

Fig. 4. MILP-based PCA Routing

TABLE I
SPEC2000 BENCHMARK CHARACTERISTICS

graph	function	#co	#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	5x5
twolf	ucxx1	353	574	590	8x8

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 shows the non-streaming benchmarks used in our experiment. The solutions to the MILPs were found using the Gnu Linear Programming Kit's [8] vesion 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 hour per MILP for standard, and 2 hours per MILP for larger cases. Typically, the solver iterates around the same minimum value for millions of iterations before being terminated, so we believe the non-optimality introduced by this time limit is negligible.

MILP problems can be solved near optimally. Thus the largest factor introducing non-optimality 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 while ESC and the greedy clustering algorithm specfically target nodes connected by edges for clustering. When comparing ESC vs 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.

When comparing the number of clusters, it is very no-

TABLE II

IMPACT OF CLUSTERING, WIRELENGTH (HOP) IS GIVEN AFTER ROUTING, EXECUTION TIME (EXEC) IS GIVEN AFTER ROUTING AS A RATIO WITH ESC CLUSTERING WIRELENGTH-ONLY DRIVEN PLACMENT AS THE BASIS, RUNTIME (CPU) IS GIVEN IN SECONDS

	random clustering				ESC clustering				greedy clustering			
bench	#clust	hop	exec	CPU	#clust	hop	exec	CPU	#clust	hop	exec	CPU
gzip	18	1984	3.63	3663	18	263	1.11	3636	18	425	0.81	3654
vpr	—	—	—	–	29	1135	0.87	3881	29	1197	0.99	3749
mcf	—	—	—	–	34	2170	0.50	36040	34	2170	0.7	36942
equake	5	206	3.04	172	5	48	1.00	97	5	91	1.00	71
parser	—	—	—	–	63	3206	0.61	4650	63	3930	0.79	36060
vortex	10	711	3.15	3623	10	181	0.89	3647	10	366	0.78	3623
bzip2	7	400	2.09	3612	7	239	0.65	158	7	207	0.78	141
twolf	—	—	—	–	44	1331	0.69	4639	44	1578	0.77	1499
ratio	2.98				0.82				0.83			

TABLE III

IMPACT OF PROFILE-BASED PLACEMENT ON EXECUTION TIME

	wirelength-driven				profile-driven			
bench	#cl	wire	exec	CPU	#cl	wire	exec	CPU
gzip	18	114	5,158	4002	18	168	5,536	3636
vpr	29	489	6,729	6658	29	625	6,250	3881
mcf	34	772	37,668	6400	34	2170	22,872	36040
equake	5	24	3,162	221	5	30	3,162	97
parser	63	1577	14,840	5241	63	1678	9,869	4650
vortex	10	63	6,009	3829	10	114	6,009	3647
bzip2	7	41	2,789	610	7	129	2,202	158
twolf	44	572	512	5153	44	747	406	4639
ratio	1.00				0.88			

TABLE IV

IMPACT OF PROFILE-BASED ROUTING ON EXECUTION TIME

	wirelength-driven				profile-driven			
bench	#cl	wire	exec	CPU	#cl	wire	exec	CPU
gzip	18	182	6,793	4002	18	263	7,550	3636
vpr	29	867	9,968	6658	29	1135	8,689	3881
mcf	34	1439	63,582	6400	34	2170	31,764	36040
equake	5	37	3,162	221	5	48	3,162	97
parser	63	3034	26,999	5241	63	3206	16,396	4650
vortex	10	107	7,726	3829	10	181	6,867	3647
bzip2	7	63	3,377	610	7	239	2,202	158
twolf	44	927	857	5153	44	1331	593	4639
ratio	1.00				0.82			

ticeable 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 noops inserted by the compiler that do not have data dependancies 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 though the number of clusters is the same the size and nodes of each cluster are different between all the algorithms.

A comparison of profile-driven versus wirelength-driven placement (Table III) and routing (Table IV) is given in their respective Tables. Our profile-driven placement algorithm outperforms the traditional wirelength-only driven algorithm by 12% in average execution time calculated before routing was done and our profile-driven router outperforms the traditional router by 18% in average execution time calculated after routing was done. 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 changes drastically from the training input set then profiling may have negative impact on performance.

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 a MILP based placement and routing algorithm that uses clustering to reduce problem complexity while still retaining solution quality.

ACKNOWLEDGMENT

This research was funded in part by DARPA PCA Program under contract #F33615-03-C-4105.

REFERENCES

[1] K. S. et. al, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proc. IEEE Int. Conf. on Computer Architecture*, 2003.

[2] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *IEEE Micro*, 2003.

[3] J. J. Granacki and M. D. V., "MONARCH: A high performance embedded processor architecture with two native computing modes," in *High Performance Embedded Computing*, 2002.

[4] M. Chu, K. Fan, and S. Mahlke, "Region-based hierarchical operation partitioning for multicluster processors," in *Proc. Programming Language Design and Implementation*, 2003, pp. 64–74.

[5] 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 *IEEE Micro*, 1992.

[6] `http://www.trimaran.org`.

[7] J. Cong and S. K. Lim, "Edge separability based circuit clustering with application to multi-level circuit partitioning," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 346–357, 2004.

[8] "GLPK (GNU linear programming) kit." [Online]. Available: `http://www.gnu.org/software/glpk/glpk.html`