Instruction Generation and Regularity Extraction For Reconfigurable Processors

Philip Brisk Adam Kaplan Ryan Kastner Majid Sarrafzadeh
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{philip, kaplan, kastner, majid}@cs.ucla.edu

ABSTRACT

The increasing demand for complex and specialized embedded hardware must be met by processors which are optimized for performance, yet are also extremely flexible. In our work, we explore the tradeoff between flexibility and performance in the domain of reconfigurable processor design. Specifically, we seek to identify regularly occurring, computation-heavy patterns in an application or set of applications. These patterns become candidates for hard-logic implementation, potentially embedded in the flexible reconfigurable fabric as special optimized instructions. In this work we present an extension to previous work in instruction generation: an algorithm that identifies parallel templates. We discuss the advantages of parallel templates, and prove the correctness of our algorithm. We introduce an All-Pairs Common Slack Graph (APCSG) as an effective tool for parallel template generation. Finally, we demonstrate the effectiveness of our algorithm on several applications' dataflow graphs, reducing latency on average by 51.98%, without unreasonably increasing chip area.

Category
1. **Compilers and Operating Systems**

General Terms
Algorithms, Performance, Theory

Keywords
Hardware Compiler, Template, Slack, Control Data-flow Graph

1. INTRODUCTION

The complexity of modern computing systems, combined with the creation of faster, smaller circuitry, has fueled the genesis of extremely small and powerful embedded systems. With the increasing design trend toward smaller and more intimate computing systems comes a need for specialization, as each of these smaller systems is more limited in functionality than a general-purpose processor. Embedded processors need not perform the same set of operations as their larger counterparts, and yet the limited number of computations they perform must be optimized for power, area, and also computational speed. Frequently, ASICs are employed in the creation of such devices, due to their traditional ability to meet these demands.

Yet, ASICs are extremely inflexible; once the device is fabricated, the functionality can never be changed. Thus, as design standards and protocols evolve, embedded ASICs must be thrown away, and new ones built to meet the new demand. This is both wasteful of good circuitry and also extremely challenging to the designers of these chips. These devices should be able to satisfy the rigid performance requirements that ASICs have classically met. Reconfigurable devices (built upon programmable logic device technology) contain logic that can be quickly and completely re-programmed. Thus, due to their inherent flexibility, reconfigurable chips have been proposed as the design platforms of specialized embedded processors. Unfortunately, there exists a tradeoff between the flexibility and performance of a system.

We desire the ability to customize a system towards the special set of tasks that it needs to perform, imbued with the ability to evolve. We believe that, given the subset of functions that such a system would need to execute frequently, it is possible to implement some of these instructions as hard (or fixed) logic blocks. Such blocks would perform better than the softer, reconfigurable parts of the device. If well chosen, they could enhance the performance of the system, providing the benefits of frequent hard-logic execution, while simultaneously affording flexibility to other (possibly subsuming) execution components.

With the goal of extracting instruction regularity, and consequently generating more complex instruction candidates for hard logic implementation, we propose a scheme whereby an application or set of representative applications would be sent to a compiler. Using the compiler's intermediate representation (IR) of a program, we can determine a candidate set of templates (instructions) to optimize. In Section 2, we will further expound upon template generation as an idea, and discuss the control dataflow graph as a compiler IR. In Section 3, we will explain a previous approach to sequential template generation, including some of its shortcomings. In Section 4, we present our algorithmic extension to template generation. In Section 5, we discuss our implementation as well as some preliminary empirical results of our instruction generation algorithm. In Section 6, we present some related work. Finally, in Section 7, we conclude with some future direction for this work.

2. TEMPLATE GENERATION

Templates are repeated occurrences of possibly interdependent nodes and edges in a dataflow graph (DFG). Each node in a DFG represents some simple operation, such as ADD or LOAD. Templates can be thought of as vertex clusters, or super-nodes, which represent instructions at the architecture level. In this paper, we define two different types of templates. Sequential templates correspond to nodes connected by directed edges in the DFG (i.e. nodes which represent a sequential execution operation). Parallel templates correspond to nodes whose operations can be scheduled for simultaneous execution. Sequential templates contain direct data dependencies (as they follow DFG edges) and thus cannot directly increase inherent parallelism in the execution. Parallel templates have no data dependencies between them, but may restrict the scheduling mechanism at lower synthesis stages.

Template generation attempts to extract regularities of simple mathematical operations on data flow graphs (DFGs). DFGs define interdependent sequences of ALU operations that occur between branching statements. DFGs are an integral piece of an intermediate compiler format known as control data flow graphs (CDFGs). CDFGs are a commonly accepted form of internal representation upon which a compiler can perform optimization and template matching. The nodes in CDFGs are basic blocks, sequences of arithmetic instructions with exactly one entrance (branch target) and exactly one exit (branching or halt instruction). An example of a CDFG is presented in Figure 1.

The template generation algorithms we discuss operate on DFGs, but do not yet extend across control nodes. Template generation is performed on the DFGs within each CDFG node, but branch-subsuming templates have not been investigated yet.

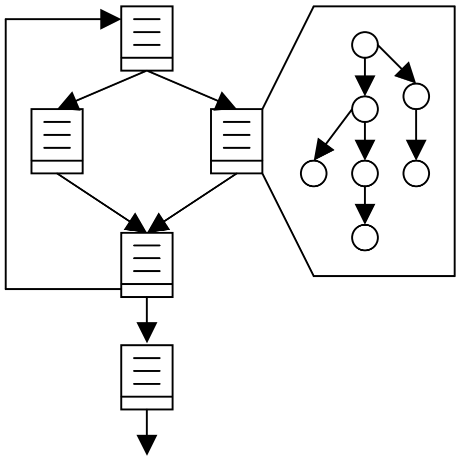

Figure 1. A Typical CDFG

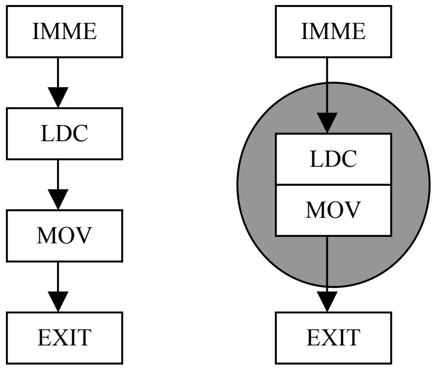

Figure 2. An example of a DAG with and without a sequential template.

3. SEQUENTIAL AND PARALLEL TEMPLATES

Sequential instruction generation was first applied to reconfigurable systems by Kastner [1]. Sequential templates attempt to extract sequential regularity in a DFG. Sequential regularity can be observed in a DFG when multiple instances of directed edges connect nodes of type A to nodes of type B (A and B correspond to simple ALU operations). Determining a set of candidate sequential templates is simple. One simply needs to examine all of the edges in a DFG and count the different types that occur. An example of a DFG with and without a sequential template is shown in Figure 2.

Since we wish to extract regularity, the edges that occur the most often will be the best candidates for sequential templates. The count for any given edge type indicates that a particular data dependency between two operations occurs frequently within the data flow of a program; however, the count alone is a misleading heuristic. Often, candidate templates will overlap one another, hence only some subset of candidate edges can legally become templates at any given time.

In Figure 3, we observe two sequential edges of type (CVT,XOR). Unfortunately, both of these edges are incident on the same XOR-node. Therefore, we cannot cluster both edges because they overlap at the XOR-node; however, we can cluster one or the other of the edges. In the future, we will consider the effects of clustering all three of these nodes into a larger template.

When it is time to actually create the template, a super-node replaces each node-edge-node combination. The super-node will have the same connectivity as the node-edge-node, and will maintain the node-edge-node information internally

Before we can cluster nodes, we must traverse the DAG and record the number of occurrences of particular edge types. Consequently, we will cluster the most frequently occurring edge type(s).

Sequential clustering iterates between the traversal and clustering phases until some stop condition is met. Without a stopping condition, all nodes and super-nodes would eventually be clustered to the point where the DAG consists of only one node, which is not a desirable outcome. There are several potential stopping conditions. Kastner [1] experimented with different stopping criteria, using the amount of graph coverage as a metric of success.

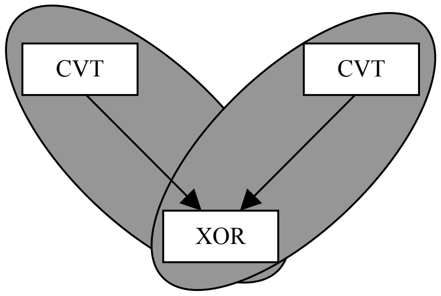

Figure 3. An example of a DAG with conflicting candidate sequential templates

4. PARALLEL TEMPLATES

Parallel templates attempt to extract forms of regularity that are free of data dependencies. Parallel regularity cannot be observed directly by examining edges in a DFG. Alternatively, one must examine DFG nodes in an order that is orthogonal to the flow of data in the DFG. An example of a DFG with a parallel template is shown in Figure 4.

Two nodes are candidates to be parallel templates if they could possibly be scheduled at the same time step by some scheduling heuristic; or equivalently, if the common slack between the nodes is greater than zero. The slack of a DFG node can be loosely described as the number time steps at which the node could be scheduled without violating data dependencies or increasing the length of the critical path of the DFG. The common slack shared by any pair of nodes in a DFG is the number of time steps at which their operations could execute simultaneously within data dependency constraints.

In order to formally determine which nodes may be scheduled together, we create an All-Pairs Common Slack Graph (APCSG) from the DFG. Every pair of nodes in the graph is considered. If the common slack between the two edges is greater than zero, an edge weighted with the common slack is added between the two nodes in question.

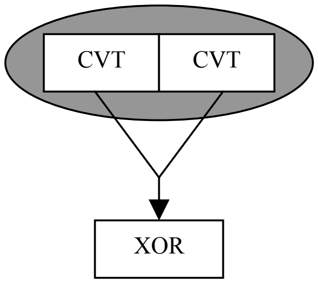

Figure 4. The DAG in Figure 3 with a parallel template.

Calculation of common slack between two nodes assumes that the graph has been topologically sorted in advance using ASAP scheduling. The level of a node is defined as the earlier possible time step at which it is scheduled during ASAP scheduling. The slack of a node may be calculated as the difference between its level sorted by ALAP scheduling and its level sorted by ASAP scheduling.

The edges in the APCSG represent the set of every pair of DFG nodes which can be scheduled at the same time step by some scheduling heuristic. Therefore, we have chosen the nodes adjacent to the set of APCSG edges to represent our candidate set of parallel templates.

A more generalized model would take note of the fact that any clique of size k in the APCSG is a set of k nodes that can be scheduled at the same time step by some scheduling heuristic. The set of APCSG edges is simply the set of all two-node cliques in the APCSG. Nonetheless, we chose only the nodes adjacent to edges in the APCSG as our candidate set of parallel templates because the clique problem is NP-complete. Examining the costs and benefits of the clique finding approach is promising future work.

To determine which types of parallel nodes should be clustered, we count the number of APCSG edges between each node type, taking into account the common slack values. Then, we cluster the corresponding DFG nodes whose APCSG edges have the highest count.

An example of a DFG and its corresponding APCSG are shown in Figures 5 and 6.

In an analogous manner to the algorithm for sequential template generation, the parallel clustering algorithm selects the most frequently occurring edge type in the APCSG. Unlike the sequential methodology, however, common slack is also used to determine exactly which type of edges should be used for contraction. For each type of edge that appears in the APCSG, a weighted sum is taken.

$$\text{Weight(parallel edge type } e) = \sum_{i \in \text{instances}(e) \text{ in APCSG}} \text{APCSG\_Weight}(i)$$

Once the weighted sums have been computed, we cluster edges of the type that has the greatest weighted sum. Among all edges of the chosen type, we will first cluster those with the largest common slack values. Once again, we justify this heuristic by arguing that clustering nodes with large common slack values will minimize the flexibility that we lose after clustering the nodes. This leads to greater amounts of regularity that can be extracted in future iterations of the algorithm. This makes it an intuitively favorable heuristic.

Initially, we prefer to cluster those edges between nodes with large common slack values. This helps to reduce the amount of slack in the DFG that is lost as a result of clustering the nodes. By similar logic, this heuristic reduces the number of edges that must be removed from the APCSG. Hence, it preserves parallelism in the DFG.

In the case of our example in Figures 5 and 6, the most frequently occurring weighted edge type is (MUL, MUL), which occurs three times with weighted sum of 4. The edge with weight 2 is chosen for clustering. The resulting DFG and APCSG are shown in Figures 7 and 8.

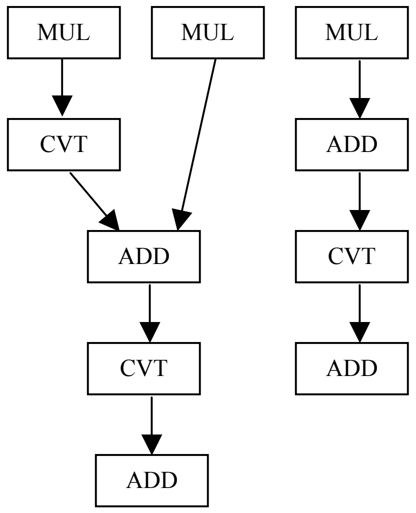

Figure 5. An example DFG that will be used to illustrate the creation of the APCSG.

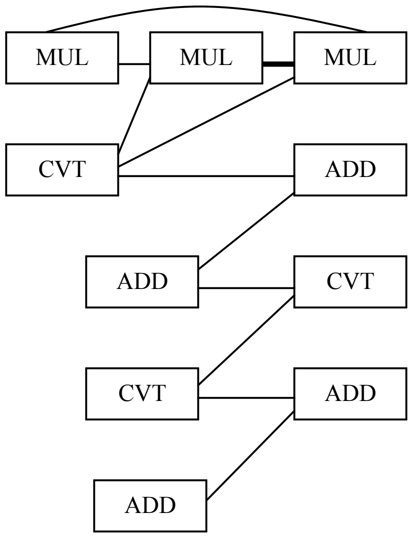

Figure 6. The APCSG. All edges have weight 1, except for the bold edge, which has weight 2.

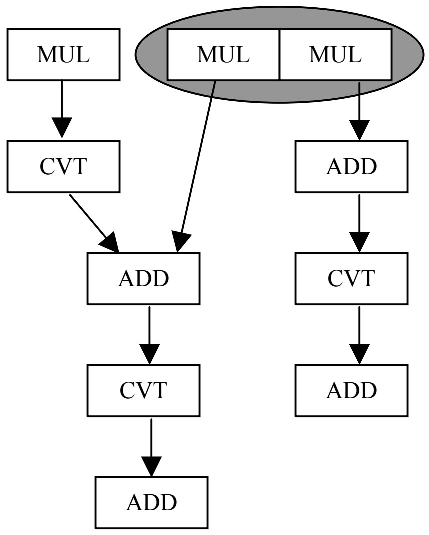

Figure 7. The updated DFG after one round of parallel clustering.

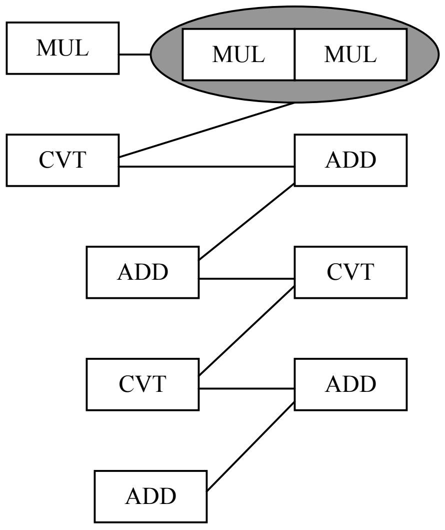

Figure 8. The APCSG after one round of parallel clustering.

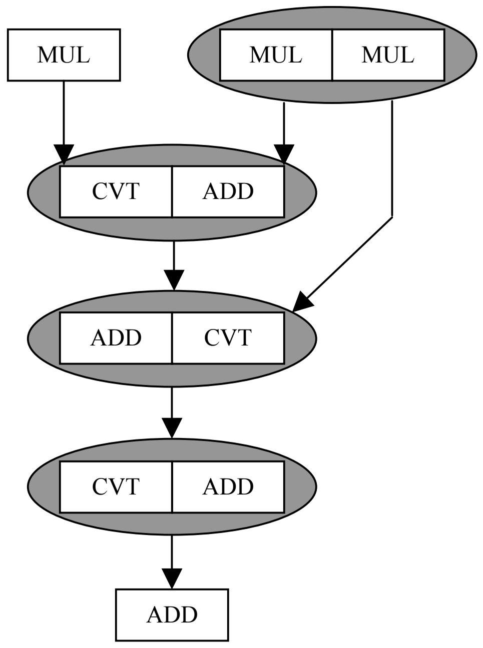

Figure 9. The updated DFG after two rounds of parallel clustering.

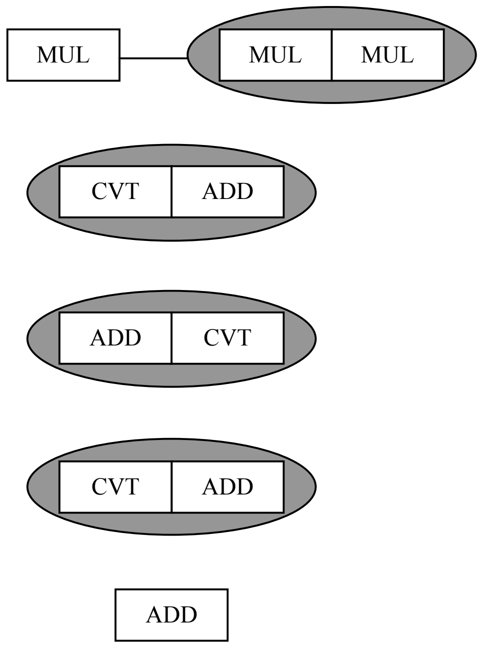

Figure 10. The APCSG after two rounds of parallel clustering.

The most frequently occurring edge type in Figures 7 and 8 is (CVT, ADD), so the corresponding nodes are chosen for clustering. Figures 9 and 10 show the resulting DFG and APCSG after one final iteration of clustering.

4.1 Combining Sequential and Parallel Template Generation

The processes of determining candidate sets of sequential and parallel templates are inherently different. Sets of sequential candidates can be deduced by examining edges in the original DFG; whereas sets of parallel candidates can be found by examining the edges in the APCSG. Candidate sets for sequential and parallel templates can be determined independently of one another, but they can be treated in the same way. If we use the weighted sum heuristic, we can combine parallel and sequential template generation into a single algorithm. At present, we equate number-of-occurrences of a given edge type with its summed slack values. (In the future, an empirically-driven modification to this cost function may be considered.)

The algorithm for simultaneous sequential and parallel template generation is shown in Figure 11. The profiling functions determine the most frequently occurring edge types in the DFG and APCSG respectively. The dispatch_clustering function determines whether the nodes being clustered are sequential or parallel, and then dispatches the information to the appropriate clustering function.

The complexity of construction of the APCSG is $O(V^2)$ because each pair of vertices in the DFG must be examined. The complexity of the profiling functions are $O(E_{dfg})$ and $O(E_{apcsg})$ respectively because each edge need only be examined once. The number of iterations of the while loop depends purely on the stopping heuristics; however, in the worst case, only one pair of nodes will be clustered during each iteration until there are no edges left. Therefore, the overall complexity of the template generation algorithm is $O(E_{dfg}^2 + E_{apcsg}^2)$.

```
1. Given a labeled DFG G(V,E)
2. G' ← construct_APCSG(G)
3. #C is a set of edge types
4. C ← {}
5. While not stop_conditions_met(G)
        a. C ← profile_graph(G) ∪
           profile_graph(G')
        b. dispatch_clustering(G, G', C)
```

Figure 11. Algorithm for Simultaneous Sequential and Parallel Template Generation

4.2 DAG Isomorphism

An important issue arises when comparing templates to one another to determine regularity. We wish to find, at every stage, the number of occurrences of a given edge type. It is mandatory to check both nodes of each edge for equivalence. If both nodes are equal to those of the type we wish to locate, then we have found another occurrence of this edge type.

As clustering commences, the vertices attached to each edge may become harder to compare. These vertices may be super-nodes, combinations of two or more original nodes with an internal directed acyclic graph (DAG) representation of the original node set. Our approach to DAG isomorphism must also consider the type of operation represented by each node. All isomorphic patterns must match node types as well as nodes and edges.

In general the test for isomorphism cannot be applied efficiently for directed acyclic graphs. Although DAG isomorphism has never been proven NP-complete, all proposed solutions to the problem possess exponential running time. We thwarted this problem in our work by integrating a polynomial-time approximation of DAG isomorphism into our implementation, via the University of Naples' VFLib Graph Matching Library [2]. Although this approximation is not perfect (i.e. it will miss some DAGs that are isomorphic to one another) it is acceptable for our purposes.

5. EXPERIMENT AND RESULTS

We implemented our algorithms on top of the Stanford SUIF compiler [3]. Building upon some of the modifications made by Kastner for sequential template construction [1], we added a full implementation of the APCSG and its generation, as well as the code to successfully merge parallel and sequential templates. Finally, we added the higher-level heuristic that performed the combination of sequential and parallel clustering. Our template sizes were restricted to five internal nodes, and our algorithm terminated when the total number of super-nodes (clustered vertices) in the DFG was less than half of the original number of vertices.

Our initial goal was to determine how template generation would affect the general scheduling of instructions, regardless of whether the target of compilation is a super-scalar pipelined architecture or the synthesis of new hardware. Although application synthesis is not the goal of the compiler, a high-level synthesis tool could perform scheduling of the resulting clustered DFGs.

Specifically, our experiment is designed to determine the scheduling latency of our generated DFGs (intuitively analogous to the time of execution on a powerful processor). We compare this latency to the scheduling latency of the original (non-clustered) DFG. Assuming that latency is generally improved by the addition of special blocks of logic to execute regular instructions, we furthermore wish to explore the impact that these clustering decisions will make on chip area. Specifically, a thorough exploration of the latency/area tradeoffs of clustering is required in order to evaluate our methods.

In order to simulate the results of our algorithm, we compiled and generated instructions for four programs: an image convolution algorithm [4], DeCSS (the decryption of DVD encoding) [5], the DES encryption algorithm [6], and the Rijndael AES encryption algorithm [7]. These algorithms are typical candidates for industrial hardware implementation (as cameras, DVD players, and embedded encryption devices must perform these operations). Additionally they are computationally intensive, leading to generally large DFGs which are benign to regularity extraction. From each compiled CDFG of the programs, four representative DFGs were selected for scheduling. The scheduling algorithm we used has been described in detail in [8], and is comparable to the state-of-the-art in the research community. The resulting latency of each scheduled DFG (both with and without clustering) is recorded in Table 1. In Table 2, we record both the decrease in latency and the increase in FPGA area that resulted from our clustering algorithm.

Clearly, clustering of DFGs reduces the number of total instructions, and increases the potential to execute frequently occurring sets of parallel operations. This directly improves the schedule of the application DFGs, demonstrating latency improvement by as much as 76.19% on our largest DFG (the first basic block of the DES encryption algorithm: 150 nodes). For every DFG scheduled, latency was improved by at least 25%, a surprisingly good figure. Additionally, the FPGA area increased an average of 21.55% (maximally 150% in some smaller DFGs). Occasionally, even decreased area was realized via clustering, presumably due to improved utilization of regular specialized components. Overall, the average latency improvement (51.98%) shadowed the area gains (average 21.55%), especially on larger, more complex DFGs.

		No Clustering	Sequential and Parallel Clustering
Convolve	Node 1	10	5
	Node 2	6	4
	Node 3	8	2
	Node 4	8	6
DeCSS	Node 1	11	6
	Node 2	10	3
	Node 3	56	31
	Node 4	21	9
DeS	Node 1	84	20
	Node 2	59	24
	Node 3	20	11
	Node 4	18	11
Rijndael AES	Node 1	24	15
	Node 2	56	32
	Node 3	17	6
	Node 4	6	2

Table 1. Latency Measurements for Each Scheduled DFG (in CPU Cycles)

		Size of original DFG (nodes)	% Latency Decrease	% FPGA Area Increase
Convolve	Node 1	20	50.00	66.67
	Node 2	13	33.33	-4.55
	Node 3	17	75.00	31.25
	Node 4	19	25.00	4.17
DeCSS	Node 1	21	45.45	150.00
	Node 2	13	70.00	-12.50
	Node 3	121	44.64	25.00
	Node 4	55	57.14	37.50
DeS	Node 1	150	76.19	-5.88
	Node 2	122	59.32	23.96
	Node 3	55	45.00	15.38
	Node 4	43	38.89	4.17
Rijndael AES	Node 1	38	37.50	20.91
	Node 2	105	42.86	33.00
	Node 3	46	64.71	-6.25
	Node 4	8	66.67	-38.10
	Averages:	52.875	51.98	21.55

Table 2. Latency Reduction and Area Increase for DFGs with Template Generation

6. RELATED WORK

Instruction selection (in the context of code generation) is the fundamental technique used by a compiler to map its intermediate code representation to a target machine's set of operations. The best-known selection algorithm is an optimal dynamic programming solution, which was first devised by Aho and Johnson [9], extending the work of Sethi and Ullman on code generation for expression trees [10]. The goal of these works was to minimize the number of total program steps (or operations performed), especially those operations performed on registers. The target architectures of their machines were known in advanced and the generation of new templates was a lengthy and complicated procedure, requiring the interface of hardware and systems-software designers. Our work is a rethinking of the instruction selection process, attempting to create the hardware and its software simultaneously.

Template generation via clusters of primitive operations has been explored before in [11, 12, 13]. These techniques are used to identify regular sequential sets of operations for use in ASIP design or high-level synthesis. To the best of the authors' knowledge, this is the first work that utilizes slack calculations on dataflow graphs in order to select parallel clusters of nodes.

PipeRench (a fully reconfigurable, pipelined FPGA architecture) utilized regularity extraction to reduce circuit area and increase performance [14]. However, their templates were hand optimized to produce beneficial results. Cadambi and Goldstein limit the form of template they consider to single output templates with a bounded set of inputs, reasoning that inputs/outputs must be bounded in order for their macros to remain routable. However, in the architecture we discuss, these instructions are implemented as small hard-logic blocks integrated into the reconfigurable fabric. These blocks can be given additional routing resources. Therefore, the restrictions provided by the authors need not be considered in our work. The authors suggested that profiling may be beneficial for small granularity FPGAs, but no supporting experiment was provided in their work to support this suggestion.

Regularity extraction has many applications in the CAD domain, including hierarchical scheduling [15], reduction of data-path complexity and improved design quality [16], system level partitioning [17], and power reduction [18].

Rao and Kurdahi [17] discussed template generation for clustering (at the system level) using the first fit bin filling heuristic. Later, Cadambi and Goldstein [19] proposed single output template generation with a bottom-up approach. Both methods restrict templates by size and number of inputs/outputs. Our algorithms are flexible enough to support such a restriction (although currently we merely impose a size constraint).

IMEC's Cathedral Project [20] used a signal flow graph rather than a CDFG to perform clustering. However, their investigation was somewhat similar to ours in spirit. The Cathedral Project investigated DSP applications at the high-level synthesis stage. Their data path was composed of Abstract Building Blocks (ABBs) (instructions available from a given hardware library). A collection of many ABBs was referred to as an application specific unit (ASU). IMEC's algorithm attempted to identify ASUs that could be executed with the best performance via manual clustering of the graph into more compact operations (similar to our template construction). However, unlike IMEC's work, our template generation algorithms are automated. Their results showed that reduction of critical path length as well as reduction of interconnect were both achievable via this method.

One of the most motivating investigations of performance gain via template matching was demonstrated in Corazao et al [21]. Much like the traditional compiler template matching algorithms, their project assumed a given library of highly regular templates. These templates could substitute for the comprising operations during the high-level synthesis stage, leading to critical path minimization. If some parts of a template were not needed, these portions were allowed to go unused (leading to a partial template matching). Their experiment resulted in large performance gains with a small area increase. Although many optimization techniques were tried as part of their strategy, template selection was described as having the largest positive performance impact.

Compton and Hauck's Totem Project [22] seeks to automate the generation of reconfigurable architectures customized for a limited set of applications. During placement and routing, they map coarse-grained components to a one-dimensional data path axis. Whereas our algorithm is given an application as high-level specification, their input is a set of architecture netlists, which are transformed into a physical design while simultaneously targeting improved routing flexibility and decreased area.

7. CONCLUSION AND FUTURE WORK

In this work we have presented template generation as a solution to the problem of regularity extraction. In particular, we have extended previous work on template generation to include parallel templates, which offer both instruction level parallelism and a previously unobserved form of parallel regularity. We have introduced the All Pairs Common Slack Graph (APCSG) as a data structure on which parallel templates can be found, and have described and verified an algorithm to perform simultaneous sequential and parallel instruction generation. Our results demonstrate large performance gains on computationally intensive algorithms with minimal FPGA area increases.

In the future, we intend to parameterize our algorithm, enabling it to produce different instructions given different system constraints (such as maximum area increase). This will allow a new level of hardware awareness in our compiler, allowing it to improve latency only when it is physically reasonable to do so. Additionally, we wish to explore DFG inlining and translations to other intermediate representations, which will provide us with a more global view of the application, as well as help us to determine other forms of instruction regularity.

8. REFERENCES

[1] R. Kastner, S. O. Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems," Proceedings of the International Conference on Computer-Aided Design, 2001.

[2] P. Foggia, C. Sansone, M. Vento. "An Improved Algorithm for Matching Large Graphs," the 3rd IAPR-TC15 Workshop on Graph-based Representations, Ischia, 2001.

[3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," IEEE Computer, December 1996.

[4] Oja E. and Karhunen, J, "Recursive Construction of Karhunen-Loeve Expansions for Pattern Recognition Purposes," Proc. of the 5th International Conference on Pattern Recognition. Miami Beach, Florida, USA, Dec. 1-5, 1980, pp. 1215-1218.

[5] Gallery of CSS Descramblers. http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/

[6] National Bureau of Standards, NBS FIPS PUB 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard," U.S. Department of Commerce, April 1981.

[7] J. Daemen and V. Rijmen, "The Block Cipher Rijndael," Smart Card Research and Applications, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296.

[8] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "A Super-Scheduler for Embedded Reconfigurable Systems," Proceedings of the International Conference on Computer-Aided Design, 2001.

[9] A.V. Aho and S.C. Johnson, "Optimal Code Generation for Expression Trees," Journal of the ACM, vol. 23, pp. 488-501, July 1976.

[10] R. Sethi and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," Journal of the ACM, vol. 17, pp. 715-728, October 1970.

[11] J Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs," in Proceedings of the 7th International Symposium on High-Level Synthesis, pp. 10-16, May 1994.

[12] C. Liem, T. May, P.G. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," Proceedings of the European Design & Test Conference, pp. 31-37, February 1994.

[13] O. Bringmann and W. Rosenstiel, "Cross-Level Hierarchical High-Level Synthesis," Proceedings of the Design Automation and Test in Europe, 1998.

[14] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, pp. 70-77, 2000.

[15] L. Tai, D. Knapp, R. Miller, and D. MacMillen, "Scheduling Using Behavioral Templates," Proceedings of the Design Automation Conference, 1995.

[16] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," Proceedings of the International Symposium on Field Programmable Gate Arrays, 1998.

[17] D. S. Rao and F. J. Kurdahi, "On Clustering for Maximal Regularity Extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, 1993.

[18] R. Mehra and J. Rabaey, "Exploiting Regularity for Low-Power Design," Proceedings of the International Conference on Computer-Aided Design, 1996.

[19] M. Kahrs, "Matching a Parts Library in a Silicon Compiler," Proceedings of the International Conference on Computer-Aided Design, 1986.

[20] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications," Proceedings of the Design Automation Conference, 1991.

[21] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey, "Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, 1996.

[22] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation," Proceedings of the Symposium on FPGAs for Custom Computing Machines Conference, 2001.