# Instruction Generation for Hybrid Reconfigurable Systems

**Ryan Kastner** **Seda Ogrenci-Memik** **Elaheh Bozorgzadeh** **Majid Sarrafzadeh**

**Computer Science Department**
**University of California, Los Angeles**
**Los Angeles, CA 90095**
**{kastner,seda,elib,majid}@cs.ucla.edu**

## Abstract

*In this work, we present an algorithm for simultaneous template generation and matching. The algorithm profiles the graph and iteratively contracts edges to create the templates. The algorithm is general and can be applied to any type of graph, including directed graphs and hypergraphs. We discuss how to target the algorithm towards the novel problem of instruction generation and selection for a hybrid (re)configurable systems. In particular, we target the Strategically Programmable System, which embeds complex computational units like ALUs, IP blocks, etc. into a configurable fabric. We argue that an essential compilation step for these systems is instruction generation, as it is needed to specify the functionality of the embedded computational units. Additionally, instruction generation can be used to create soft macros – tightly sequenced pre-specified operations placed in the configurable fabric.*

## 1 Introduction

Regularity extraction attempts to find common sub-structures (templates) in one or a collection of circuits (graphs). There are many applications for regularity extraction, including, but not limited to, scheduling during logic synthesis [1], system-level partitioning [2] and FPGA mapping and placement [3]. We aim to build a general profiling technique for simultaneous template generation and matching, which is applicable to any task that uses a directed labeled graph. We target the generation and matching algorithm towards instruction generation and selection, though the methods we present are general enough to be applied to any regularity extraction problem represented by a labeled digraph.

A novel use of template generation and matching is instruction generation during compilation for hybrid (re)configurable systems. *Hybrid (re)configurable systems* contain some kind of computational unit, e.g., ALUs, IPs or even traditional processors embedded into a reconfigurable fabric. *Instruction generation* combines basic operations into larger, more complex operations. The new operations may be quite complex, involving hundreds of basic operations, e.g. an FFT or filtering operation, or something less complex like a multiply-add-accumulate (MAC) operation. The complex operations must occur frequently to overcome limitations of tightly sequencing the operations and the (possibly fixed) area that they occupy in the fabric.

In this paper, we focus on instruction generation for the *Versatile Programmable Blocks (VPBs)* -- embedded hard-wired blocks that perform complex instructions -- of the *Strategically Programmable System (SPS)* [4]. The SPS architecture combines memory blocks and VPBs into a fine-grained reconfigurable fabric. Additionally, regularity extraction can indicate which operations are important to optimize as soft macros implemented on a fine-grained reconfigurable fabric of the SPS. The selected sets of operations should occur frequently within the application. A system architect may choose to hand optimize these operations to decrease power, increase performance and/or minimize the silicon footprint. Cadambi and Goldstein show that soft macro regularity extraction is beneficial for their PipeRench architecture [5].

In the next section, we formalize the problem of template matching and generation. Section 3 proposes an algorithm for simultaneous template generation and matching using graph profiling and edge contraction. In Section 4, we present experimental results. We conclude in Section 5.

## 2 Problem Formulation

### 2.1 Template Matching

*Regularity* refers to the repeated occurrence of computational patterns e.g. multiply-add patterns in an FIR filter and bi-quads in a cascade-form IIR filter. A *template* refers to an instance of a regular computational pattern.

We model an algorithm, circuit or system using a digraph *G(V,E)*. The nodes of the graph correspond to an instance of basic computational units. Examples of node types are add, multiply, subtract, etc. Each node has a label consistent with the type of operation that it performs. The edges of a graph model the dependencies between two operations. For instruction generation, the graph under consideration is a dataflow graph. We consider labeled digraphs in this work as we are targeting instruction generation, which use dataflow graphs; dataflow graphs can be sufficiently modeled using labeled digraphs.

There are two general problems associated with template matching:

**Problem 1:** *Given a directed, labeled graph* G(N, A), *a library of templates, each of which is a directed labeled graph* $T_i$(V,E), *find every subgraph of* G *that is isomorphic to* $T_i$.

This problem is essentially equivalent to the subgraph isomorphism problem simplified due to the directed edges. Even with these simplification the general directed subgraph isomorphism problem is NP-complete [6].

**Problem 2:** *Given an infinite number of each set of templates $\Omega = T_1, \ldots, T_k$ and an overlapping set of subgraphs of the given graph G(N,E) which are isomorphic to some member of $\Omega$; minimize k as well as $\Sigma x_i$ where $x_i$ is the number of templates of type $T_i$ used such that the number of nodes left uncovered is the minimum.*

We want to minimize both the number of distinct templates that are used in the covering while minimizing the number of instances of each template. Additionally, we want cover as many nodes as possible. This problem is a fusion of the graph covering and the coin changing problems. It differs from the graph covering as it allows multiple instances of template in it's covering. The coin changing problem tries to find the minimum number of coins to produce exact change; this is similar to minimizing the number and types of templates to cover the graph.

## 2.2 Template Generation

Most previous work have assumed that the templates were given as an input. However, this may not always be the case; an automatic regularity extraction algorithm must develop it's own templates.

Consider instruction selection for hybrid reconfigurable architectures, specifically the SPS. First, we must determine the functionality of the VPBs (hard-wired embedded blocks) during the SPS architecture generation phase. Template generation is a necessary step here. Secondly, unlike traditional processors where the instructions (templates) are fixed according to the ISA, it is possible to arrange fine-grained reconfigurable fabric to perform virtually any combination of basic operations. Hence, instruction templates are not fixed in fine-grained reconfigurable architectures. Template generation is a necessary step for the hybrid (re)configurable architecture generation and compilation.

The custom instructions should be generated to maximally cover the dataflow graph. In essence, the compiler must perform instruction generation and selection, equivalently template generation and matching.

## 3 An Algorithm for Simultaneous Template Generation and Matching

In this section, we present an algorithm for template generation and matching which iteratively clusters nodes based on profiling. During the clustering, we generate templates and find a cover using the templates. The algorithm starts by profiling the graph to determine the frequency of node and edge types. Based on the most frequently occurring edges, clustering is performed. An overview of the algorithm is given in Figure 1.

```
1   Given a labeled digraph $G(V, E)$
2   # $C$ is a set of edge types
3   $C \leftarrow \varnothing$
4   while stop_conditions_met($G$)
5       $C \leftarrow$ profile_graph($G$)
6           cluster_common_edges($G, C$)
```

**Figure 1 Overview of clustering-based algorithm for template generation and matching**

## 3.1 Algorithm Description

The algorithm starts by calling the function *profile_graph*, which traverses the graph to record the occurrence of a particular edge. It returns the edge types ordered corresponding to the frequency of their appearance in the graph.

The function *cluster_commom_edges* takes the labeled digraph and the edge type frequencies and performs node clustering based on edge contraction. Given two vertices $v_1$ and $v_2$, *contraction* removes $v_1$ and $v_2$, replacing them with a new vertex $v$. The set of edges incident on $v$ are the union of the set of edges incident to $v_1$ and $v_2$. Edges from $v_1$ and $v_2$ with mutual endpoints, e.g. $e_1 = (v_1, x)$ and $e_2 = (v_2, x)$, may or may not be merged into one edge; we merge them but we must remember that these two edges exist to preserve the sanity of the logic. We further discuss this and other issues concerning clustering later.

Finally, the function *stop_conditions_met* is called to possibly halt the algorithm. The function returns "true" if the algorithm has generated a sufficient amount of templates and/or the graph sufficiently covered. Often, we wish stop when once a certain number of templates are generated. Another possible stopping condition is when the generated templates cover every vertex of the graph. Most likely, the stopping condition function should be tailored to the particular application for template generation and matching. We discuss the stopping conditions we choose for instruction selection in the next section.

Figure 2 demonstrates two passes of the algorithm. The initial graph (Figure 2a) is profiled and the edge type (*,*) is chosen for clustering. You can see that there are many conflicting choices for edge contraction (Figure 2b). We discuss how to resolve these conflicts later. Edges 2 and 3 are clustered to form a supernode (Figure 2c). The next round of profiling chooses to contract the edge (*, {*:*}) where {*:*} is the supernode created in the previous pass. Edge contraction occurs to create a super-supernode. The algorithm stops having generated a template and a covering using these templates (Figure 2d).

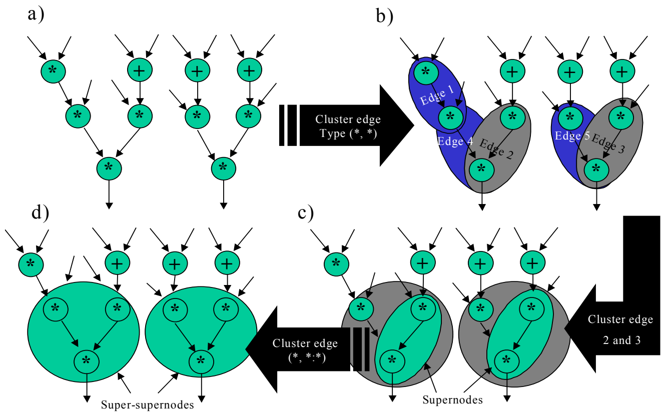

**Figure 2 Contraction of edges to create supernodes. The supernodes correspond to templates.**

Edge contraction essentially creates a supernode from two nodes. The supernode must hold the DAG of the operations that it implements in order to realize the templates that it is generating; we call this the (super)node's *internal DAG.*

Every time we create a new supernode, we generate a new template corresponding to that new node. That template is the internal DAG of the supernode. It is possible that identical templates are generated through separate sequences of clustering. Therefore, we cannot identify the template based on its sequence of edge contractions; we must consider the supernode's internal DAG. A graph isomorphism algorithm (e.g. Gemini [7]) is needed to identify whether two templates (generated through a different sequence of edge contractions) are identical.

## 3.2 Finding an Optimal Covering

Choosing the sequence of edges to contract can greatly affect the quality of the solution. Consider the graph in Figure 2. The best cover consists of one template consisting of two multiply operations feeding into another multiply (as demonstrated in Figure 2d). But, if we contracted Edge 1 instead of Edge 2 and 3, we would not have been able to achieve this cover. This dilemma has haunted graph covering algorithm makers for a long time; there is no known exact method to avoid such ill-fated decisions. We employ a heuristic based on maximum independent set of a conflict graph.

A conflict graph is an undirected graph $G_c(V_c,E_c)$. There is a vertex $v \in V_c$ for each possible instance of a template in the labeled digraph $G$; $G$ is the graph that we are performing template generation and matching. There is an edge $e = (v_1, v_2)$ between the vertices $v_1$ and $v_2$ if two different instances of a template share a common node in $G$. Since we are trying to maximize the number of nodes that are covered using the minimum number of templates, we want to cover as many nodes possible at each step. This is accomplished by taking the maximum independent set of nodes in the conflict graph. The *Maximum Independent Set (MIS)* is the largest subset of vertices *S* of a graph *G* such that no two vertices of *S* are adjacent.

**Theorem 3.1** *Given a graph G(V,E) to cover with a template T, the template instance assignment corresponding to the nodes from the maximum independent set of the conflict graph* $G_c(V_c,E_c)$ *gives an optimal covering where optimality is defined as a covering the maximum number of vertices in V.*

Since the vertices of the conflict graph correspond with covering of nodes, finding the MIS of the conflict graph $G_c$ gives a covering of the maximum number of nodes. MIS is NP-complete [6]; fortunately the simple *minimum-degree algorithm* [8] has been shown to give good results. We use that algorithm to find a good set of edges to contract.

## 4 Experimental Results

### 4.1 Experimental Setup

To test our template generation and matching algorithm, we implemented a hybrid (re)configurable system compiler front-end on top of the SUIF2 compiler system [9]. SUIF is a well-known intermediate format (IF) that is used heavily in industry and academia. It compiles C/C++/Fortran source code into a high-level IF. We used the Machine-SUIF [10] back end to create a low-level IF representation, i.e. a Control Flow Graph (CFG). From there, we implemented a pass to convert the CFG to a *Control Dataflow Graph (CDFG)*. Our template generation and matching algorithm was performed over all the dataflow graphs of a CDFG.

We look at the applications from the MediaBench test suite [11]. From these applications, we selected a set of files that implement DSP functions. Table 1 presents the characteristics of the selected DSP functions.

| Benchmark | C File | Description |
|---|---|---|
| mpeg2 | motion.c | Motion vector decoding |
| mpeg2 | getblk.c | DCT block decoding |
| adpcm | adpcm.c | ADPCM to/from 16-bit PCM |
| epic | convolve.c | 2D general image convolution |
| jpeg | jctrans.c | Transcoding compression |
| jpeg | jdmerge.c | Color conversion |
| rasta | fft.c | Fast Fourier Transform |
| rasta | noise_est.c | Noise estimation functions |
| gsm | gsm_decode.c | GSM decoding |
| gsm | gsm_encode.c | GSM encoding |

**Table 1 MediaBench test files**

### 4.2 Results

We ran the template generation and matching algorithms on the test files. For each test file, a set of templates was generated and a covering was produced using the generated templates. Templates were generated on the nodes that performed arithmetic operations. The stopping condition of the algorithm depended on the frequency of the most often occurring edge. If the edge type occurred less than *x*%, the algorithm completes. We call this the *cut-off percentage.*

We varied the cut-off percentage to measure the tradeoff of number of generated templates vs. percentage of the graph covered by those templates. As the cut-off percentage increases, the number of generated templates decreases and fewer nodes are covered. As it decreases to 0, the algorithm generates a larger number of templates and covers more nodes, but the additional templates that are generated may only cover a few additional nodes.

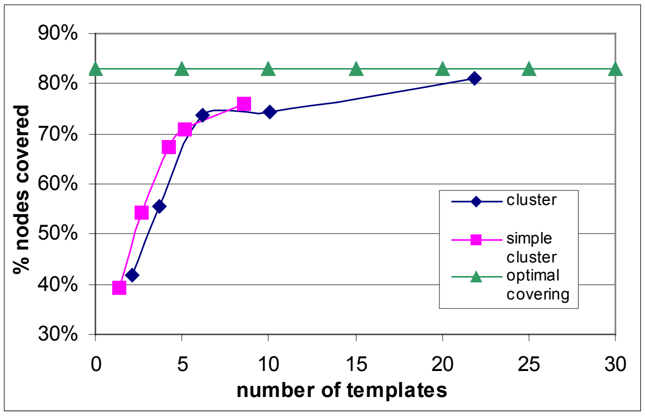

**Figure 3 Comparison of clustering techniques**

The "cluster" line in Figure 3 plots the average number of templates generated and the percentage of the graph covered using the generated templates. By varying the cut-off frequency we produced the points of the graph. A cut-off frequency of zero will cover every node by creating templates that occur a small number of times (including singleton templates). Sometimes a graph cannot be completely covered by templates, as an arithmetic node is isolated in a dataflow graph (CFG node) by itself. In the benchmarks that we consider, the "optimal" covering is a covering of 83% of the nodes i.e., on average, 17% of the nodes are isolated. You can see that in order to generate an optimal covering the algorithm generates an average of 21.9 templates.

The slope of the line ($\Delta$(% coverage)/ $\Delta$templates) gives much intuition into the amount of coverage you get by generating additional templates. When the number of templates is small (less than 5) the slope is large, meaning that adding another template gives you a large amount of additional graph coverage. As the number of templates increases, the slope decreases. It is interesting to note that the slope dramatically reduces around 5 templates. It seems to suggest that using five templates is a good number for covering the benchmarks.

During our experiments, we noticed that the number of operations (node) per template is small. We tried restricting the edge contraction so that only templates with 2 nodes would be generated. We called this "simple" clustering. The results using this clustering scheme were plotted in Figure 3. The points were generated by varying the cut-off percentage in a similar manner as the previous experiment. As you can see, the results mimic those of the "complex" clustering technique. The main difference is that the simple technique cannot achieve an optimal covering like the complex technique. But, in order to achieve an optimal covering, the complex technique generated a large amount of templates. Many of these generated templates covered a limited number of nodes - a poor solution. Therefore, a template generation and matching which limits the templates to 2 nodes gives a solution with similar quality as the complex algorithm.

Additionally, we noticed the types complex templates varied widely across all the applications. Therefore, if we wanted to generate one "generic" system for all the benchmarks, e.g. a system of DPS applications, that we examined, there would be a large number of templates and each application would use only a small subset of those templates. On the other hand, we found that there was much less variation of template types when we used the simple templates.

| Operation | MediaBench file name | | | | |
|---|---|---|---|---|---|
| | motion | jdmerge | getblk | gsm_dec | jctrans |
| ADD | 50.3% | 84.6% | 44.5% | 29.6% | 84.6% |
| MUL | 36.3% | 13.8% | 24.0% | 22.4% | 13. 8% |
| Template Coverage | | | | | |
| MUL-MUL | 0.0% | 0.0% | 1.3% | 0.0% | 0.0% |
| ADD-ADD | 14.5% | 9.1% | 3.2% | 3.6% | 9.1% |
| ADD-MUL | 0.0% | 0.4% | 0.6% | 0.0% | 0.4% |
| MUL-ADD | 36.3% | 13.0% | 21.5% | 22.4% | 13.0% |

**Table 2 Coverage using simple add and multiple template combinations**

To further explore this phenomenon, we looked at simple template combinations using add and multiply combinations – the two most frequently occurring arithmetic node types across all the benchmarks. Table 2 shows the results of the coverage using the 4 add/multiply sequences as individual templates. In the table, the notation OP1-OP2 denotes that the template consists of the two operations with an edge {OP1, OP2}.

We can gather a lot of information using these simple templates. For example, the sequence of operations deviates from probability theory as the sequence MUL-ADD is found with much greater frequency than ADD-MUL. Probabilistically, these sequences should occur in the same proportion. Additionally, it shows that the MUL-ADD and ADD-ADD sequences could be implemented as a VPB or macro for DSP applications as it is widely used across all the applications. In summary, we presented evidence that templates can be limited to simple, two operation sequences while achieving good coverings using a small number of templates. We believe that this is due to the structure of the CDFGs.

In general, the dataflow graphs of the CDFGs are not deep and the actual number of arithmetic operations per dataflow graph is not large enough to encourage templates with a large cardinality. These are two well-known phenomena and are the source of problems in exploiting parallelism for VLIW processors. Therefore, we believe that hybrid (re)configurable systems must leverage techniques from the VLIW domain such as predicated execution and hyperblock construction in order to realize larger template cardinality.

## 5 Conclusion

In this work, we addressed the problems of template generation and matching. We proposed an algorithm to perform simultaneous template generation and matching. Our algorithm generates templates by profiling the graph and clustering common edges. To our knowledge, this is the first algorithm to consider template generation without input/output restrictions.

Template generation is a relatively new and essential problem for compilation to configurable systems. Template generation can be used to create macros, which are tightly coupled sequential operations that are placed in the same vicinity in a configurable fabric. Furthermore, the macros are ideal candidates for hand optimization. Additionally, template generation can be used to specify the functionality for pre-placed ASIC blocks (VPBs) in hybrid (re)configurable systems like SPS.

We developed the front-end compiler for a hybrid (re)configurable system. Using DSP benchmarks, we showed that full-blown template generation is unnecessary as simple templates – templates with a sequence of two operations – create a graph covering with similar quality to more complex templates. This suggests that advanced compiler techniques such as predicated executation and hyperblock construction are needed in order to efficiently utilize large templates.

In the future, we plan to study the effect of that predicated execution and hyperblock on template generation. Also, we intend to develop a complete back-end of a retargetable compiler for hybrid (re)configurable systems.

## References

[1] T. Ly, D. Knapp, R. Miller and D. MacMillen, *Scheduling using Behavioral Templates*, Design Automation Conference, 1995.

[2] D. S. Rao and F. J. Kurdahi, *On Clustering for Maximal Regularity Extraction*, IEEE Trans. on Computer-Aided Design, Vol. 12, No. 8, August, 1993.

[3] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal and R. Gupta, *A General Approach for Regularity Extraction in Datapath Circuits*, International Conference on Computer-Aided Design, 1998.

[4] S. Ogrenci-Memik, E. Bozorgzadeh, R. Kastner and M. Sarrafzadeh, *Strategically Programmable Systems,* Reconfigurable Architecture Workshop, 2001.

[5] S. Cadambi and S. C. Goldstein, *CPR: A Configuration Profiling Tool*, Symposium on Field-Programmable Custom Computing Machines, 1999.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[7] C. Ebeling and O. Zaijicek, *Validating VLSI Circuit Layout by Wirelist Comparison*, International Conference on Computer-Aided Design, 1983.

[8] M. M. Halldórsson and J. Radhakrishnan, *Greed is Good: Approximating Independent Sets in Sparse and Bounded-degree Graphs*, ACM Symposium on the Theory of Computing, 1994

[9] R. Wilson *et al.*, *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, ACM SIGPLAN Notices, Dec. 1996.

[10] M. D. Smith, *Extending SUIF for Machine-dependent Operations*, SUIF Compiler Workshop, 1996

[11] C. Lee, M. Potkonjak and W. H. Maggione-Smith, *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*, IEEE/ACM International Symposium on Microarchitecture, 1997.