Instruction Generation for Hybrid Reconfigurable Systems

R. KASTNER
University of California, Santa Barbara
and
A. KAPLAN
University of California, Los Angeles
and
S. OGRENCI MEMIK
University of California, Los Angeles
and
E. BOZORGZADEH
University of California, Los Angeles

Future computing systems need to balance flexibility, specialization and performance in order to meet market demands and the computing power required by new applications. Instruction generation is a vital component for determining these tradeoffs. In this work, we present theory and an algorithm for instruction generation. The algorithm profiles a data flow graph and iteratively contracts edges to create the templates. We discuss how to target the algorithm towards the novel problem of instruction generation for hybrid reconfigurable 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 reconfigurable macros – tightly sequenced pre-specified operations placed in the reconfigurable fabric.

Categories and Subject Descriptors: [Hardware]: System Design
General Terms: Algorithms, Design
Additional Key Words and Phrases: FPGA, High-level Synthesis, Reconfigurable Computing

---

1. INTRODUCTION

Computational devices are becoming more complex. The number of transistors on a single die – dictated by Moore's Law – is increasing exponentially. This allows a *system-on-chip* – a variety of different computing devices interacting as a complete system on a single die. An additional benefit of Moore's Law is the decreasing cost of computations leading to a ubiquity of embedded systems. Much like a system-on-chip, these embedded

This research was supported by the National Science Foundation, Xilinx and Fujitsu.
Authors' addresses: R. Kastner, Department of Electrical and Computer Engineering, University of California, Santa Barbara. 93109; A, Kaplan, S. Ogrenci Memik, E. Bozorgzadeh, Computer Science Department, University of California, Los Angeles, 90095.

systems are a complex interaction of many different computational devices, embedded within a larger entity e.g. a car, telephone, building, etc.

With the proliferation of computing systems comes a need for specialization; each use of the system is tailored for a specific set of applications. For example, a digital system embedded within a cellular phone will encounter DSP-type applications. It most likely needs to perform operations like analog to digital conversions (and vice-versa), FFT, and filtering. Therefore, if we customize the embedded system to such operations, we gain increased performance, power/energy reduction and smaller silicon footprint. This tends towards the use of ASICs for such systems.

Yet, an ASIC is extremely inflexible; once the device is fabricated, the functionality cannot be changed. For example, if a new cellular communication standard emerges, we must throw away our phone and buy a new one customized for the new standard. However, if a new standard appears and the embedded system is flexible, we can change the functionality of the system to handle the migration from one standard to another. Another increasingly important trend is time to market. The initial market share of a product that is released first is inherently larger than that of a product that is released much later. In fact, we are seeing that time to market is becoming vital to the success of the product. The public accepts new products at a fanatical pace. Products once took 10+ years to gain consumer acceptance (e.g. television, radio). Now, products permeate the market in under a year (e.g. DVD, MP3 players). Both of these trends accentuate the need for flexible devices like a general purpose processor. The more applications/standards that the device can service, the more companies will use it, as they want to reach the market quickly and service as many markets as possible.

We have conflicting forces pushing digital systems in two seemingly separate directions. On one hand, computing systems must be specialized to meet the performance, power, energy, and area constraints. In this regard, ASICs are the answer. On the other hand, time to market and flexibility constraints push for general purpose systems. Obviously, there is a tradeoff between the flexibility (general purpose) and performance (application specific) of the system. We need methods to allow us to perform the tradeoffs between these two metrics. We must be able to customize a system towards the tasks that it will most likely perform and give it the flexibility to adapt over the course of its lifetime.

Customized instructions are one way to explore the tradeoff between customization and flexibility. They approach the problem through the customization of a general purpose device. If we know the *context* – a set of applications that will likely run on a

system, we can look for commonly occurring, computational sequences within the various applications of the context. These customized instructions can be optimized for high-performance, low energy/power consumption and/or small area. For example, it is well known that the computational sequence, multiply-accumulate (MAC), occurs frequently in DSP applications. The MAC unit would be an ideal candidate for a customized instruction when the context is DSP applications. In this paper, we develop a systematic method for customized instruction generation and explore the theoretic aspects of *instruction generation* – the process of finding commonly occurring computational patterns within a context.

Customized instructions can serve to optimize two general purpose devices – the processor and the FPGA. *Application specific instruction set processors (ASIPs)* take a small processor core and add customized instructions to service the specific context of the applications. The PICO project [Schreiber, et al. 2000] aims to automatically generate the customized instructions based a specific application. They use a VLIW core and generate nonprogrammable hardware accelerators (NPA), which are akin to systolic arrays. The interface between the NPAs and the processor core is automatically synthesized. The user is responsible for identifying the customized instructions in the form of loop nests. Another product, Tensilica's Xtensa processor [Gonzalez 2000], takes the customized instructions as an input in the form of a *hardware description language (HDL)* called the Tensilica Instruction Extension (TIE) Language. It incorporates the instruction into a compiler allowing the user to execute the instruction through an intrinsic function. Our instruction generation algorithms can serve the Xtensa and PICO frameworks to automatically find the customized instructions.

The FPGA is another general purpose computing device, albeit quite different from a general purpose processor. The FPGA has the benefit of adapting its architecture directly to the application that it implements. Data intensive applications running on an FPGA can achieve up to 100x increased performance as compared to the same application running on a processor [Athanas and Abbott 1995, Gokhale, et al. 1991, Peixin, et al. 1999, Sidhu, et al. 1999]. This mainly comes from the ability to customize at the architecture level. The architecture of an application running on an FPGA is completely flexible, whereas the processor architecture is fixed.

Yet, the architecture of an FPGA is still tailored for the general case. Adding *macros* to the architecture could customize the FPGA. Macros are hard or soft reconfigurable computational sequences. A *hard macro* is a fixed ASIC core embedded into the fabric of the FPGA. The embedded multipliers of the Xilinx Virtex series are an example of a

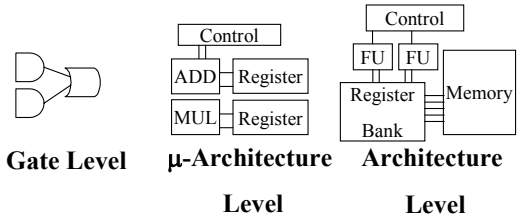
hard macro. A *soft reconfigurable macro* is a sequence of computations that are implemented as a fixed entity on the FPGA fabric. Examples of soft reconfigurable macros are the components of the Xilinx CoreGen library.

In this sense, reconfigurable architectures are moving away from reconfiguration exclusively at the gate level and moving towards a *hybrid reconfigurable architecture*. Hybrid reconfigurable architectures contain reconfigurability at multiple levels of the *computational hierarchy* (see Figure 1). The computational hierarchy is the level of abstraction that computations may be implemented. One level of the computational hierarchy is the gate or Boolean level. At this level, every computation is built up from the Boolean (gate) level computations. The FPGA is an example of a device that functions at this level. A device can also be reconfigured at the microarchitecture or architecture level. These levels of computational hierarchy have coarser basic computational units. A basic unit at the microarchitecture level is on the level of an arithmetic function. PipeRench [Goldstein, et al. 2000] and RaPiD [Ebeling, et al. 1996] are examples of this type of reconfigurable system. At the architecture level, the basic unit of computation is more coarse grained, for example the RAW project [Taylor, et al. 2002].

Reconfigurability at the various levels of the computational hierarchy gives many tradeoffs in terms of flexibility, reconfiguration time, performance, area and power/energy consumption. A fine-grained reconfigurable device (gate level) is extremely flexible; it can implement any application. However, the flexibility comes at a cost. The routing architecture must allow a connection from any part of the chip to any other part of the chip. Switchboxes are used to enable this sort of flexibility. The switchboxes are composed of many transistors to enable a flexible routing. Compared to a direct connection, it is apparent that switch boxes add much overhead to the area, delay (performance), and power/energy consumption. Furthermore, the implementation of an arithmetic unit (e.g. an adder) on a fine-grained reconfigurable device consists of programming each gate of the arithmetic unit. Since the gates are designed to be extremely flexible – you can implement any Boolean function on any gate – the arithmetic unit will be large, slow and power/energy hungry as opposed to the same arithmetic unit that is designed specifically for implementing that function, as is the case for a device that is reconfigurable at the microarchitecture level. On the other hand, the area, delay, power/energy consumption of implementing a Boolean function favors a gate level reconfigurable device. If we implement a one bit “and” function on a device that is reconfigurable at the architectural level, the function will be over designed. It will

implement an “and” instruction and 31, 63 or 127 “and” operations will be unnecessarily performed, depending on the size of the instruction. There are benefits for reconfigurability at various levels of the computational hierarchy. A hybrid reconfigurable architecture allows us to mix and match these levels to tailor to the applications at hand.

Figure 1: A comparison between three levels of the computational hierarchy. The gate level is the most flexible with the architecture level being the least flexible. The architecture level has the fastest reconfiguration time. The performance, area and power/energy consumption depend on the type of operation being implemented.



Reconfigurability	Bit	Byte	Instruction (32 –128 bits)
Basic Unit of Computation	Boolean Operation (and, or, xor)	Arithmetic Operation (add, multiply)	Functional Operation (ALU, MAC)
Communication	Connections through switchboxes	Bundles of wires, registers	Bus, memory

Examples of hybrid reconfigurable systems include Garp [Callahan, et al. 2000], which couples a MIPS-II processor with a fine-grained FPGA coprocessor on the same die, the Strategically Programmable System (SPS) architecture [Ogrenci Memik, et al. 2001] combines memory blocks, Versatile Programmable Blocks (VPBs) – embedded ASIC blocks that perform complex instructions – into a LUT-based fabric. Many other academic projects can be called a hybrid reconfigurable system, for example Dynamically Programmable Gate Array (DPGA) [DeHon 1996] and Chimaera [Hauck, et al. 1997].

In addition, several industrial projects fall into the category of hybrid reconfigurable systems. One example is the Virtex-II devices from the new Xilinx Platform FPGAs, which embed high-speed multipliers into their traditional LUT-based FPGAs. Also, the CS2112 Reconfigurable Communications Processor (RCP) from Chameleon Systems,

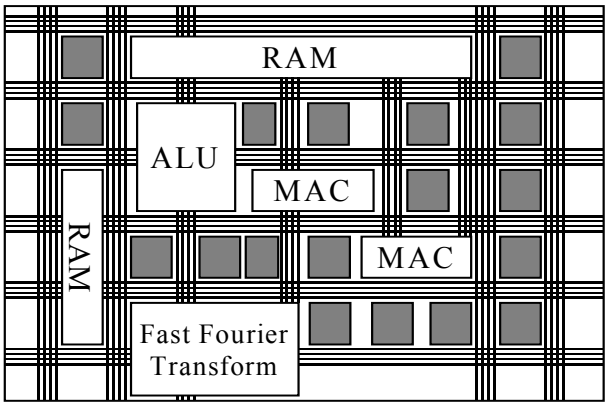
Inc. contains reconfigurable fabric organized in slices, each of which can be independently reconfigured.

To understand how instruction generation works in hybrid reconfigurable systems, we consider the SPS project. SPS consists of VPBs embedded into a LUT-based fabric. It is targeted towards a specific context. An example of a specific SPS architecture is shown in Figure 2. Because the VPBs are hard macros, implementing an operation on them as opposed to on the reconfigurable fabric will give lower power and energy consumption. Additionally, the VPBs require less time to program. Hence, the SPS architecture can be reconfigured faster than an FPGA. Furthermore, the performance of the operation on the VPB will be much better than the performance of the same operation on the reconfigurable fabric. However, we must carefully consider the type of functionality for the VPB. If the applications of the context never use the VPBs, then they are wasting space on the chip.

A designer of an SPS architecture can specify the functionality of the VPBs towards the targeted context. The designer must consider the types of operations that make up the applications of the context. The operations must occur frequently. Furthermore, the operations must give performance or other benefits (e.g. reduced power consumption) when implemented as VPB as opposed to on the reconfigurable fabric. There are many tradeoffs to consider when choosing the functionality of the VPB. A tool to determine these tradeoffs would help the designer pick the functionality. Ideally, the tool would automatically determine the functionality of the VPBs based on the given context. This is exactly the problem of instruction generation.

Instruction generation is also useful for the generation of soft reconfigurable macros. Soft reconfigurable macros act as a "black box" to a designer. The input, output and functionality of the soft reconfigurable macro are given to the designer, but the actual implementation of the macro on any specific reconfigurable architecture is abstracted away. In this sense, the terms soft reconfigurable macro and soft reconfigurable IP (intellectual property) are synonymous. A soft reconfigurable macro is customized for every reconfigurable architecture. For example, the Xilinx CoreGen library has IPs like decoders, filters, memories, etc. Each of these functions is customized for the architecture on which it runs. The CoreGen components are generated based on what Xilinx believes their customers will use. Also, the functionalities of the CoreGen IPs are well-known entities. Instruction generation is useful to find unknown, irregular computational patterns that are not immediately apparent from looking at the code of the applications in a context.

Figure 2: Example of the Strategically Programmable System (SPS) – a hybrid reconfigurable system. Functional units are embedded within a reconfigurable fabric. This SPS architecture would be specific to the DSP context.



Soft reconfigurable macros give many benefits. They allow the designer to work at a higher level of abstraction. Instead of dealing with basic arithmetic operations like addition, multiplication, etc, the designer can implement the application using function or block level structures. Matlab works at this level and is extremely popular in the signal processing community. Additionally, the soft reconfigurable macros can be highly optimized. A person familiar with the underlying architecture can design each macro. Therefore, the macros will be more efficient than if someone who does not understand the underlying architecture implemented them or if they were designed from basic operations using the synthesis flow. Finally, the compilation time of the applications using the macros is reduced. The macros can be pre-placed and routed. Therefore, the tool or designer must only determine the location of the macros on the reconfigurable fabric. The placement and routing of the macro – an extremely time consuming task – is unnecessary.

In summary, instruction generation is an extremely important concept for context-specific architectures. Whether the underlying architecture is derived from a general purpose processor or a reconfigurable architecture, instruction generation is essential to the flexibility and performance of the system. Furthermore, instruction generation in the form of soft reconfigurable macros can reduce the time for synthesizing an application to reconfigurable architectures.

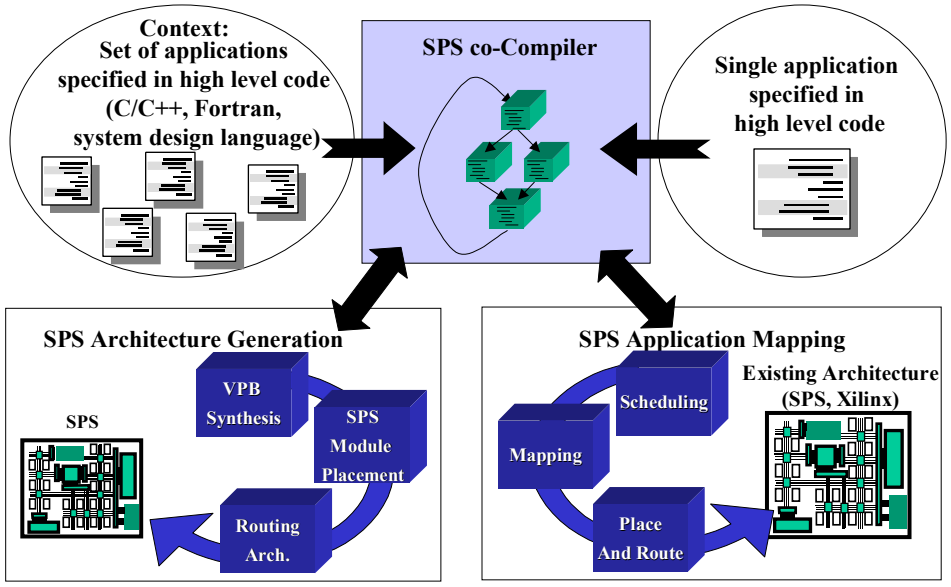
In the next section, we look at the role of instruction generation in a reconfigurable system compiler. The following section formalizes the problem of instruction generation.

We discuss the specifics of the formulation of the instruction generation problem in Section 3. Section 4 proposes an iterative, constructive algorithm for simultaneous template generation and matching using graph profiling and edge contraction. In Section 5, we present experimental results. Then, in Section 6, we discuss related work. We conclude in Section 7.

2. INTEGRATION OF INSTRUCTION GENERATION WITH A RECONFIGURABLE SYSTEM SYNTHESIS

Reconfigurable system synthesis has two different uses. One flow is architecture generation and another flow for application mapping. Figure 3 gives a high level overview for both of these flows. We use the SPS co-compiler as a representative reconfigurable system compiler.

Figure 3: Two flows for the SPS system. The flow on the left is the SPS architecture generation flow. The application mapping flow is on the right. The co-compiler interfaces with both flows.



In both flows, the first task is to translate each of the applications of the context or the single application into a form that is suitable to interface with architecture generation and application mapping, respectively. The applications are given in a high level language. For example, the applications could be written in C/C++, Fortran or some other system design language, like SystemC [Grotker 2002], SpecC [Gajski, et al. 2000] or Esterel

[Edwards 2002]. We must choose an intermediate representation (IR) for the co-compiler. We use the *control data flow graph (CDFG)* as the intermediate representation.

A CDFG is a directed, labeled graph with data nodes corresponding to operations (addition, multiplication, shift) and control nodes that dictate the flow of the program. Control nodes allow branches and loops. The edges between the nodes represent control and data dependencies. The CDFG offers several advantages over other models of computation. The techniques of data flow analysis (e.g. reaching definitions, live variables, constant propagation, etc.) can be applied directly to CDFGs. Also, many high-level programming languages (e.g. Fortran, C/C++) can be compiled into CDFGs with slight modifications to pre-existing compilers; a pass converting a typical high-level IR into control flow graphs and subsequently CDFGs is possible with minimal modification. Therefore, we can leverage the front-end of many existing compilers for our reconfigurable system compiler.

On the other hand, CDFGs only have the ability to describe instruction level parallelism. In order to specify a higher level of parallelism, another model of computation (MOC) must be used. CDFGs could be embedded into another MOC – one that can describe a higher level of parallelism. For example, we could embed CDFGs into finite state machines (FSM). Lee's *charts [Girault, et al. 1999] do something similar; they embed synchronous data flow graphs into a FSM.

Instruction generation has a role in both the architecture generation flow and the application mapping flow. In the architecture generation flow, instruction generation determines the functionality of the hard macros – the VPBs in SPS. During application mapping, instruction generation is needed to determine soft reconfigurable macros.

There are many other tasks in the architecture generation and application mapping flows. For example, we must determine the exact placement of the VPBs, their interface with the reconfigurable fabric, the routing architecture, the exact number of each type of VPB and so on. We focus on generating the VPB functionality and refer the interested reader to our other papers on these subjects [Bozorgzadeh, et al. 2002, Bozorgzadeh, et al. 2002, Kastner, et al. 2002, Kastner, et al. 2001, Ogrenci Memik, et al. 2001, Ogrenci Memik, et al. 2001].

3. PROBLEM FORMULATION

The instruction generation problem is an example of *regularity extraction*. 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, system-level partitioning and FPGA mapping and placement. In our case, the templates we extract will be our instructions and the collection of graphs is the set of applications – the context – that we wish to target our system towards.

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 directed labeled graph.

3.1 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 data flow graph.

It should be noted that systems/circuits must often be modeled by hypergraphs. A *hypergraph* is like an ordinary graph, but each *hyperedge*, connects multiple vertices instead of two as in a normal digraph. Extending our algorithms to consider hypergraphs is fairly straightforward.

We consider labeled digraphs in this work as we mainly target instruction generation, which use compiler data flow graphs; data flow 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(V, E), *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 [Garey and Johnson 1979].

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(V,E) *which are isomorphic to some member of* Ω*; 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.*

An example of these two problems is given in Figure 4. First, we must determine the exact location of all of the templates as stated in Problem 1. Once we have found every occurrence of the template, Problem 2 selects a set of templates that maximizes the covering of the graph using the templates.

We want to minimize both the number of distinct templates that are used in the covering and the number of instances of each template. Additionally, we want to 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 its 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.

The classical compiler problem of instruction selection falls into the realm of template matching. We are given the templates or instruction mappings corresponding to a directed, labeled graph of the program more commonly known as the IR. Instruction selection is directly related to Problem 2, with possibly additional objectives e.g. minimize runtime of the code, size of the code, etc.

3.2 Template Generation

Until this point, it was assumed that the templates were given as an input. However, this may not always be the case; an automatic regularity extraction algorithm must develop its own templates.

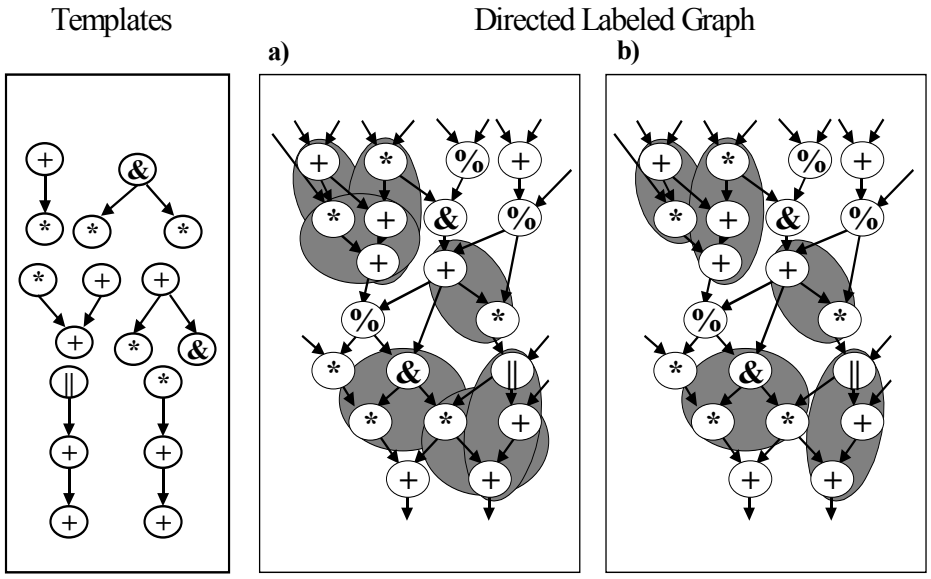
Consider instruction generation for hybrid reconfigurable architectures. The instructions (templates) for traditional processors are fixed according to the target architecture. Since we are dealing with hybrid reconfigurable architectures, the instructions are not fixed. It is possible to arrange the reconfigurable fabric to perform virtually any combination of basic operations. Therefore, the instruction templates are not fixed in reconfigurable architectures and template generation is a necessary step for the hybrid reconfigurable architecture generation and compilation.

The configurable fabric allows the designer to implement custom instructions as well as perform any fixed instructions on more traditional embedded processing units. The custom instructions should be generated to maximally cover the data flow graph. In

essence, the compiler must perform instruction generation and selection, equivalently template generation and matching.

Additionally, template generation is useful for creating macro libraries for both ASIC and FPGA architectures [Cadambi and Goldstein 1999]. Also, template generation is needed for effective system-level partitioning [Rao and Kurdahi 1993].

Figure 4 a) shows the subgraphs isomorphic to the given templates as described in Problem 1. Figure 4 b) selects a non-overlapping set of these subgraphs such that the covering of vertices is maximized



4. 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 for 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 5.

4.3 Algorithm Description

The algorithm starts by calling the function *profile_graph*, which traverses the graph to record the number of occurrences of every edge type. It returns the edge types ordered corresponding to the frequency of their appearance in the graph. Since the edges are not

labeled, the two node types (head and tail) that an edge connects identify its type. If there are N distinct node types, the number of distinct edge types is $O(N^2)$.

Figure 5: Overview of clustering-based algorithm for template generation and matching

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

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. Edges between v_1 and v_2 are removed to eliminate self-loops; self-loops must be eliminated when considering acyclic graphs as they introduce a cycle.

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 is sufficiently covered. Without a stopping condition, the clustering process would continue until there is only one node. However, it is unlikely that we want this to happen. Often, we wish to stop 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.

Theorem 4.1 *One iteration of the clustering-based algorithm takes time $O(|E|)$ on the graph $G(V,E)$.*

Proof: One iteration spans steps 4 – 6 in Figure 5. The function profile_graph looks at every edge of the graph to determine the frequency of the edge types which takes $O(|E|)$ time. Using a table of size $O(|L|^2)$ where L is the number of distinct node labels (one entry for each possible edge type), we can determine the edge type in constant time for

each edge by looking at the nodes adjacent to the edge. Furthermore, we know that $O(|L|) \leq O(|E|)$. Since we are only concerned with the most common edge type, we must only know of the edge type with largest number of occurrences. We can keep track of this when we are incrementing the edge type of the current edge. If the number of occurrences of the most common edge type is less than the number of occurrences of the edge type currently being incremented, then we set the most common edge type as the current edge type; there is no need to sort the edges. The act of clustering (edge contraction) takes constant time for each edge we wish to contract. We must also determine the edges that we wish to contract. A simple method to choose the set of edges to contract is to consider each edge and contract the edge as if it is possible i.e. the previous edge contractions did not include a vertex that is incident to this edge. This takes $O(|E|)$ time but is not optimal. We consider other methods in Section 4.3 that are provable optimal, but may increase the runtime of an iteration. We assume that the halting condition of the algorithm takes constant time, which is true if our halting condition depends on the number of templates (constant time check) generated or if it halts once the graph is sufficiently covered (constant time check). Therefore, the total runtime is $O(|E|)$. □

Figure 6: Contraction of edges to create supernodes. The supernodes correspond to templates.

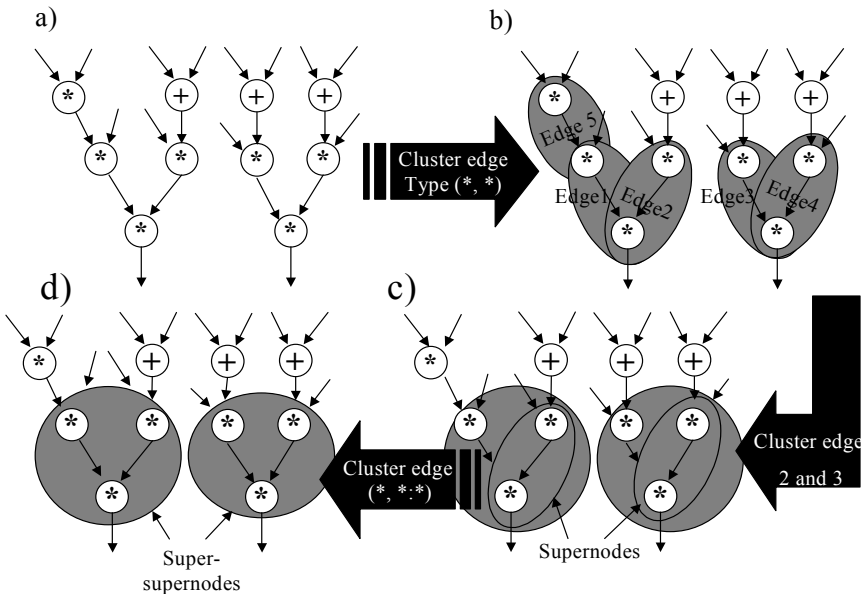


Figure 6 demonstrates two passes of the algorithm. The initial graph (Figure 6a) is profiled and the edge type (*,*) is chosen for clustering. You can see that there are many conflicting choices for edge contraction (Figure 6b). We discuss how to resolve these conflicts later. Edges 2 and 4 are clustered to form a supernode (Figure 6c). 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 6d).

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 is needed to identify whether two templates (generated through a different sequence of edge contractions) are identical.

4.2 Quick Rejection Graph Isomorphism

We developed a graph isomorphism algorithm for DAGs that quickly rejects dissimilar graphs while determining their isomorphism. The algorithm is sketched in Figure 7. Our algorithm is an extension of Gemini [Ebeling and Zajicek 1983]; the Gemini algorithm iteratively colors and recolors vertices according to vertex invariants until every vertex contains a unique color equivalently each vertex is in a unique partition. An *invariant* is a property of a graph that does not depend on its presentation. More formally, an invariant is a function F such that $F(G_1) = F(G_2)$ if G_1 is isomorphic to G_2. The algorithm approaches the problem by finding a canonical label (coloring or partitioning) for each graph and then compares the labels of the graphs. If the labels are equivalent, then the graphs are isomorphic. The Gemini algorithm is used to solve the general graph isomorphism problem. We use additional invariant properties and checks related to labeled digraphs e.g. level information, to create a better initial coloring, which should decrease the number of iterations.

First, the algorithm does a simple check to verify if the two graphs G_1 and G_2 have the same edge and vertex cardinality. Then, the graphs are sorted in reverse topological

order while adding level information to each vertex. The *level* of a vertex v is the minimum distance from v to $v' \in PO$ where PO is the set of primary output vertices (vertices with outdegree = 0). An example of a graph with level information is shown in Figure 8.

We use the level information as an initial color for the vertices (lines 6 – 10). At this point, we can compare the number of edges between levels as an addition check. Next, the vertices are iteratively recolored according to their colors and the colors of the adjacent edges (lines 12 – 16). This continues until each vertex of the graphs has a unique color. If the color for each and every vertex in G_1 matches a color for a distinct vertex in G_2, then the graphs are isomorphic (line 17). The coloring procedure is described in further detail in the Gemini graph isomorphism algorithm.

If we didn't care about runtime, we could solely use Gemini. But often this is overkill as it is rare that we create isomorphic templates. Therefore, the initial checks will most often quickly determine that the internal DAGs are non-isomorphic. Since we need to perform isomorphism checks after every iteration, it is essential to the overall runtime of the algorithm that we have a fast graph isomorphism algorithm.

Figure 7: An overview of the labeled DAG isomorphism algorithm

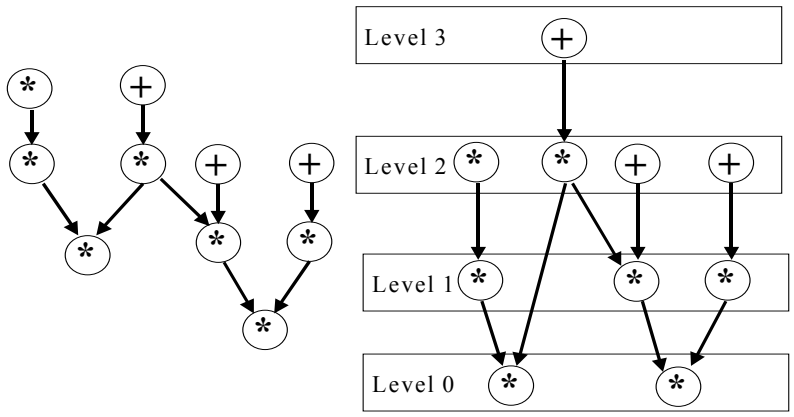
```
1   Given labeled digraphs G1(V1, E1) and G2(V2, E2)
2   if | V1| != | V2| or | E1| != | E2|
3       return false
4   R1 = reverse_topological_order(G1)
5   R2 = reverse_topological_order(G2)
6   for each level l of R1
7       for each vertex v in level(R1, l)
8           color(v)
9       for each vertex v' in level(R2, l)
10          color(v')
11  //Now run the Gemini algorithm
12  while fully_partitioned(V1,V2) = false
13      for each vertex v in V1
14          color(v)
15      for each vertex v in V2
16          color(v)
17  return equivalent_vertex_labels(G1, G2)
```

Theorem 4.2 *The labeled DAG isomorphism algorithm is correct.*

Proof: The basis of the algorithm is using the invariant properties of a graph to create a unique color or partition for each vertex. A vertex invariant is a property of the vertex that is preserved under isomorphism. It is trivial to show that the level of a vertex is

invariant. Therefore, the initial coloring is invariant. The remainder of the algorithm iteratively colors (partitions) the vertices as stated in the Gemini algorithm, which is known to be correct. □

Figure 8: Levels of a digraph



4.3 Finding an Optimal Covering

Choosing the set of edges to contract can greatly affect the quality of the solution. Consider the graph in Figure 6. The best cover consists of one template consisting of two multiply operations feeding into another multiply (as demonstrated in Figure 6d). However, if we contracted Edge 5 instead of Edge 2 and Edge 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 locally optimal heuristic based on maximum matching.

A *matching* of a graph is any set of pairwise disjoint edges. A maximum matching of a graph is the matching with maximum cardinality. The maximum matching problem for general undirected graphs can be solved in $O(n^{2.5})$ time [Micali and Vazirani 1980].

To find the optimal set of edges to contract, we use the maximum matching of the edge induced undirected subgraph G_E of G. The edges inducing the subgraph have the most common edge type for the current iteration. 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 matching of the edge induced subgraph G_E. The edges in the maximum matching are chosen as the edges to contract.

Theorem 4.3 *Given a graph G(V,E) to cover with a template T, the template instance assignment corresponding to the edges from the maximum matching of the edge induced subgraph G_E of G gives an optimal covering where optimality is defined as a covering of the maximum number of vertices in V.*

Proof: We prove the theorem by contradiction. Assume that the covering using maximum matching of G_E is not optimal. Since the number of vertices in each template instance is the same, a larger number of template instances correspond to a better matching (covering of more vertices). If the current matching *M* is not optimal, then there must be another covering *M'* such that the number of templates used in *M'* is greater than the number of templates used in *M*. By definition of a matching, the template instances in the covering of *M'* correspond to a set of edges in G_E that have no edges between them. Therefore, we can construct a matching with a larger cardinality than the maximum matching, giving the contradiction. □

Therefore, we can use the maximum matching algorithm to find a locally optimal covering of templates for the current iteration. Using the best know algorithm for maximum matching on a general undirected graph, we can optimally solve the template covering problem of any iteration of our template generation algorithm in $O(n^{2.5})$ time.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

To test our template generation and matching algorithm, we implemented a hybrid reconfigurable system compiler front-end on top of the SUIF2 compiler system [Hall, et al. 1996]. 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 [Smith and Holloway] 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 CDFG. Our template generation and matching algorithm was performed over all the data flow graphs of a CDFG.

5.2 Target Applications

Digital signal processing (DSP) applications tend to have a large amount of parallelism between instructions. They are ideal for mapping onto reconfigurable hardware, as the instruction parallelism can be exploited for increased runtime speed compared to executing the instructions sequentially on a traditional processor. In this work, we focus on DSP functions.

We look at the applications from the MediaBench test suite [Lee, et al. 1997]. From these applications, we selected a set of files that implement DSP functions. Table 1 presents the characteristics of the selected DSP functions.

5.3 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.*

Table 1 MediaBench test files

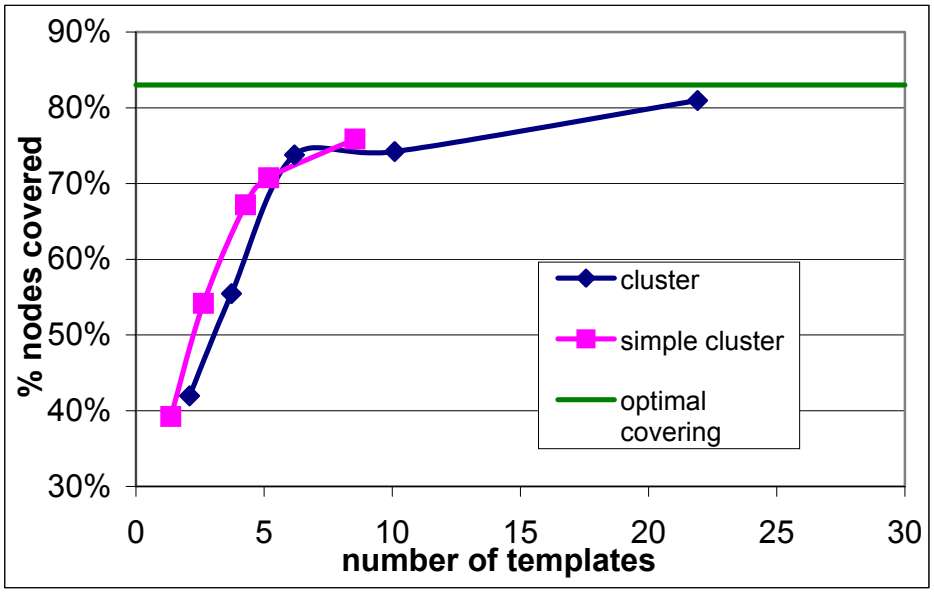
Benchmark	**C File**	**Description**
mpeg2	motion.c	Motion vector decoding
mpeg2	getblk.c	DCT block decoding
adpcm	adpcm.c	ADPCM to/from 16-bit PCM
epic	convolve.c	2D general image convolution
jpeg	jctrans.c	Transcoding compression
jpeg	jdmerge.c	Color conversion
rasta	fft.c	Fast Fourier Transform
rasta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

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.

Remember that a template refers to a sequence of operations that will reside in the same vicinity and execute in sequence. The template can be placed as a macro in the configurable fabric or be used to specify the functionality of a VPB block. Either way, we lose system flexibility in hopes to increase performance, reduce power, etc. The gain of using a template in the system is a hard parameter to quantify. It depends on a number

of factors, including scheduling, placement, routing, etc. Regardless of those other factors, a template must occur frequently in order to yield a favorable performance/power to flexibility ratio. Therefore, we aim to minimize the number of different templates while maximizing the covering of the graph(s).

Figure 9: Comparison of clustering techniques



The "cluster" line in Figure 9 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 data flow 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 (Δ(% coverage)/ Δtemplates) 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 9. As with the previous experiment, the cut-off percentage is varied to generate the different points. 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. 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, template generation and matching which limits the templates to 2 nodes gives a solution with similar quality as the complex algorithm.

Table 2: Coverage using simple add and multiple template combinations

Oper-ation	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%

Additionally, we noticed that the types of 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 DSP 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.

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 the expected probability 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[1] 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 data flow graphs of the CDFGs have a small number of levels and the actual number of arithmetic operations per data flow 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 reconfigurable systems must leverage techniques from the VLIW domain such as predicated execution [Mahlke, et al. 1995] and hyperblock construction [Mahlke, et al. 1992] in order to realize larger template cardinality.

6. RELATED WORK

While there has been a lot of work in regularity extraction, most of it focuses on template matching (similar to the graph covering problem) and not template generation.

Regularity extraction was shown beneficial in reducing area and increasing performance for the PipeRench architecture [Goldstein, et al. 2000]; PipeRench is a fully reconfigurable, pipelined FPGA. The benefits stem from the fact that the templates may be hand optimized. Additionally, the template operations are placed in the same vicinity on the chip. This reduces the interconnect delay as well as compacts the application into a smaller portion of the chip. Cadambi and Goldstein go on to show that templates lead to a decrease in area and delay for the PipeRench architecture; they suggest that profiling is beneficial for small granularity FPGAs e.g. LUT or PLA-based, though no empirical evidence is given to support this claim.

---

[1] The MUL-ADD should come as little surprise as we are profiling DSP applications and the multiply-add (MAC) instruction is a staple of the DSP instruction set.

Cadambi and Goldstein restrict their template generation to single output templates and limit the number of inputs. If the templates are going to be used as soft reconfigurable macros and placed in a configurable fabric, the number of inputs/outputs must be limited to maintain good routability. But, templates can also be used to generate the VPB functionality. Since the VPBs are ASIC blocks integrated into the reconfigurable fabric, the system architecture can place additional routing resources around VPBs to handle the additional routing needed by VPBs with a large number of inputs/outputs. Therefore, generated templates need not always have input/output restrictions.

Regularity extraction is used in a variety of other CAD applications. Templates are used during scheduling to address timing constraints and hierarchical scheduling [Tai, et al. 1995]. Data path circuits exhibit a high amount of regularity; hence regularity extraction reduces the complexity of the program as well as increasing the quality of the result [Callahan, et al. 1998, Chowdhary, et al. 1998]. System level partitioning is yet another use of regularity extraction [Rao and Kurdahi 1993]. Furthermore, proper use of templates can lead to low-power designs [Mehra and Rabaey 1996].

One of the earliest template matching works in the CAD community was by Kahrs [Kahrs 1986], wherein a greedy, bottom-up procedure for a silicon compiler is described. Keutzer [Keutzer 1987] modeled a system as a DAG and heuristically partitioned it to yield rooted trees and applied compiler techniques to test for pattern matches. Trees and single output templates were used by Chowdhary et al. [Chowdhary, et al. 1998] to cover data path circuits.

Rao and Kurdahi [Rao and Kurdahi 1993] addressed template generation for system-level clustering using the well-known first fit approach to bin filling. More recently, Cadambi and Goldstein [Cadambi and Goldstein 1999] proposed single output template generation via a constructive, bottom up approach. Both methods restrict the area and the number of pins for their templates. Our method attempts to find the best possible set of templates, regardless of area and size, though we can easily add pin and area restrictions to our algorithms. Additionally, we perform template generation and matching simultaneously.

IMEC's Cathedral Project [Note, et al. 1991] used a different model of computation in their high-level synthesis stage: instead of a CDFG they performed reductions on the signal flow graph of a DSP application. Their data path was composed of Abstract Building Blocks (ABBs), or instructions available from a given hardware library. The customized data path generated from many ABBs was referred to as an application

specific unit (ASU). Cathedral's synthesis targeted ASUs, which could be executed in very few clock cycles. This goal was achieved via manual clustering of necessary operations into more compact operations, essentially a form of template construction. Whereas our template generation and matching algorithms are automated, the definition of clusters in Cathedral was a manual operation, mainly clustering loop and function bodies. Their results demonstrated an expected reduction of critical path length as well as interconnect as a result of clustering.

One of the more encouraging cases of performance gain via template matching was investigated by Corazao et al [Corazao, et al. 1996]. Their work assumed a given library of highly regular templates. These templates could be utilized during the high-level synthesis stage in order to minimize the number of clock cycles in a circuit's critical path. In circumstances where some parts of a template were not needed, partial matching was also allowed. With partial matching, some portions of a selected template go unused. Their experimental results demonstrated large performance gains without an unreasonable increase in area. Although many optimization techniques were utilized as part of the synthesis strategy, template selection had the largest impact on overall improvement in throughput.

The Totem Project [Compton and Hauck 2001, Compton, et al. 2002] endeavors to automate the generation of custom reconfigurable architectures based on a given set of applications. Built upon the RaPiD architecture [Ebeling, et al. 1996], their optimizations are made at the placement and routing stages of synthesis, mapping coarse-grained components to a one-dimensional data path axis. Unlike our design, their input is a set of architecture netlists, which are transformed directly to a physical design while targeting the simultaneous goals of increased routing flexibility and decreased area.

7. CONCLUSION

In this work, we addressed the problem of instruction generation. We proposed an algorithm to solve the problem that performs simultaneous template generation and matching. Our algorithm generates instructions by profiling the graph and clustering common edges. Furthermore, we present some theory behind instruction generation.

Instruction generation is a relatively new and essential problem for compilation to reconfigurable systems. Instruction generation can be used to create soft reconfigurable 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 reconfigurable systems.

We developed a co-compiler for a hybrid reconfigurable 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 execution and hyperblock construction are needed in order to efficiently utilize large templates.

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

REFERENCES

ATHANAS, P.M. and ABBOTT, A.L. 1995, Real-time image processing on a custom computing platform. *Computer*, *28* (2). 16-25.

BOZORGZADEH, E., et al. 2002. Pattern selection: customized block allocation for domain-specific programmable systems. in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*.

BOZORGZADEH, E., et al. 2002, SPS: Strategically programmable system - fully automated architecture generation and application compilation, UCLA Technical Report.

CADAMBI, S. and GOLDSTEIN, S.C. 1999. CPR: a configuration profiling tool. in *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*.

CALLAHAN, T.J., et al. 1998. Fast module mapping and placement for datapaths in FPGAs. in *Proceedings of the International Symposium on Field Programmable Gate Arrays*.

CALLAHAN, T.J., et al. 2000, The Garp architecture and C compiler. *Computer*, *33* (4). 62-69.

CHOWDHARY, A., et al. 1998. A general approach for regularity extraction in datapath circuits. in *Proceedings of the International Conference on Computer-Aided Design*.

COMPTON, K. and HAUCK, S. 2001. Totem: Custom Reconfigurable Array Generation. in *Proceedings of the Symposium on FPGAs for Custom Computing Machines Conference*.

COMPTON, K., et al. 2002. Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems. in *Proceedings of the International Symposium on Field Programmable Logic and Applicaions*.

CORAZAO, M.R., et al. 1996, Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *15* (8). 877-888.

DEHON, A. 1996. DPGA utilization and application. in *Proceedings of the International Symposium on Field Programmable Gate Arrays*.

EBELING, C., et al. 1996. RaPiD-reconfigurable pipelined datapath. in *Proceedings of the Workshop on Field-Programmable Logic and Applications*.

EBELING, C. and ZAJICEK, O. 1983. Validating VLSI circuit layout by wirelist comparison. in *Proceedings of the International Conference on Computer-Aided Design*.

EDWARDS, S.A. 2002, An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *21* (2). 169-183.

GAJSKI, D.D., et al. 2000. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston.

GAREY, M.R. and JOHNSON, D.S. 1979, Computers and intractability. A guide to the theory of NP-completeness.

GIRAULT, A., et al. 1999, Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *18* (6). 742-760.

GOKHALE, M., et al. 1991, Building and using a highly parallel programmable logic array. *Computer*, *24* (1). 81-89.

GOLDSTEIN, S.C., et al. 2000, PipeRench: a reconfigurable architecture and compiler. *Computer*, *33* (4). 70-77.

GONZALEZ, R.E. 2000, Xtensa: a configurable and extensible processor. *IEEE Micro*, *20* (2). 60-70.

GROTKER, T. 2002. *System Design with SystemC*. Kluwer Academic Publishers, Boston.
HALL, M.W., et al. 1996, Maximizing multiprocessor performance with the SUIF compiler. *Computer*, *29* (12). 84-89.
HAUCK, S., et al. 1997. The Chimaera reconfigurable functional unit. in *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*.
KAHRS, M. 1986. Matching a parts library in a silicon compiler. in *Proceedings of the International Conference on Computer-Aided Design*.
KASTNER, R., et al. 2002, Compiler techniques for system synthesis optimization, UCLA Technical Report.
KASTNER, R., et al. 2001. Instruction generation for hybrid reconfigurable systems. in *Proceedings of the International Conference on Computer Aided Design*.
KEUTZER, K. 1987. DAGON: technology binding and local optimization by DAG matching. in *Proceedings of the Design Automation Conference*.
LEE, C., et al. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. in *Proceedings of the International Symposium on Microarchitecture*.
MAHLKE, S.A., et al. 1995. A comparison of full and partial predicated execution support for ILP processors. in *Proceedings of the International Symposium on Computer Architecture*.
MAHLKE, S.A., et al. 1992. Effective compiler support for predicated execution using the hyperblock. in *Proceedings of the International Symposium on Microarchitecture*.
MEHRA, R. and RABAEY, J. 1996. Exploiting regularity for low-power design. in *Proceedings of the International Conference on Computer-Aided Design*.
MICALI, S. and VAZIRANI, V.V. 1980. An O(√|V| |E|) algorithm for finding maximum matching in general graphs. in *Proceedings of the Symposium on Foundations of Computer Science*.
NOTE, S., et al. 1991. Cathedral-III: architecture-driven high-level synthesis for high throughput DSP applications. in *Proceedings of the Design Automation Conference*.
OGRENCI MEMIK, S., et al. 2001. Strategically programmable systems. in *Proceedings of the Reconfigurable Architecture Workshop*.
OGRENCI MEMIK, S., et al. 2001. A super-scheduler for embedded reconfigurable systems. in *Proceedings of the International Conference on Computer Aided Design*.
PEIXIN, Z., et al. 1999, Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *18* (6). 861-868.
RAO, D.S. and KURDAHI, F.J. 1993, On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *12* (8). 1198-1208.
SCHREIBER, R., et al. 2000. High-level synthesis of nonprogrammable hardware accelerators. in *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*.
SIDHU, R.P.S., et al. 1999. String matching on multicontext FPGAs using self-reconfiguration. in *Proceedings of the International Symposium on Field Programmable Gate Arrays*.
SMITH, M.D. and HOLLOWAY, G. An introduction to machine SUIF and its portable libraries for analysis and optimization, Division of Engineering and Applied Sciences, Harvard University Technical Report.
TAI, L., et al. 1995. Scheduling using behavioral templates. in *Proceedings of the Design Automation Conference*.
TAYLOR, M.B., et al. 2002, The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, *22* (2). 25-35.