

Adapting Static Single Assignment for Hardware Compilation

Ryan Kastner, Elaheh Bozorgzadeh, Seda Ogrenci Memik and Majid Sarrafzadeh

{kastner, elib, seda, majid}@cs.ucla.edu

Computer Science Department

University of California, Los Angeles

Los Angeles, CA 90095

Abstract

This paper describes methods for synthesizing the internal representation of a compiler into a hardware description language; a process often referred to as hardware compilation. We present a framework for this transformation including methods to control the path of execution and ways to deal with the data communication. We show how static single assignment (SSA) is useful to reduce the amount of data communication in the hardware. In some applications, SSA reduces the data communication by 70 fold. We demonstrate that the placement of Φ -nodes by current SSA algorithms is not optimal in terms of minimizing data communication. We develop a new SSA algorithm for Φ -node placement that considers data communication. We show that our algorithm reduces the data communication for some applications as much as 20% as compared to the best-known SSA algorithm – the pruned algorithm.

1 Introduction

The increasing complexity of hardware design has brought about many challenges. One of the main challenges lies in the specification of an application and subsequently, the mapping of that application into hardware. Both of these challenges focus on moving the abstraction level of hardware design to the application level – a realm where the software community excels. On the other hand, the increased complexity of hardware allows more functionality to be implemented in hardware. Applications what we were once relegated to software because of their complexity can now be implemented in hardware. Consequently, the traditionally distinct line between the hardware and software communities has blurred.

The level of abstraction in hardware design has moved to point of programming languages. There are many initiatives towards an application-level *hardware description language (HDL)* including SystemC [1] and SpecC [2]. Typically, these HDLs have a C-like semantics with additional constructs to describe parallelism and ease the *hardware compilation* process – the mapping of an application into hardware.

Hardware compilation has a variety of benefits. First and foremost, programmers can describe applications at a much higher level of abstraction. Describing an application in a low-level HDL – behavioral or *register transfer language (RTL)* description – is an arduous task; it corresponds to programming in assembly. Obviously, if the hardware compiler can produce a low-level description on par with a human coded description, the benefits

are enormous. Furthermore, emerging *systems-on-chip (SOC)* consist of a variety of components. An application programmer (or automated partitioning engine) must be able to determine the tradeoffs between allocating computations on the various SOC components. A application-level hardware language would streamline this process.

In this paper, we look into the process of hardware compilation. More specifically, we look at the process of automatically mapping an application onto a micro-architecture. We describe a methodology that takes a traditional programming language (C, C++, Fortran) and produces a HDL description. We describe methods of minimizing the interconnect of the HDL description using *static single assignment (SSA)*. We show an inherent deficiency of SSA and describe a new form of SSA that is better suited for hardware.

In the next section, we give background material related to our research. Section 3 discusses our framework for hardware compilation. We show how SSA is useful to minimize interconnect in the hardware in Section 4. Furthermore, we point out a fundamental shortcoming of traditional SSA and develop a new SSA algorithm to overcome this limitation. Section 5 presents experiments to illustrate the effect of these algorithms to minimize data communication. We discuss related work in Section 6 and provide concluding remarks in Section 7.

2 Preliminaries

2.1 Control Data Flow Graphs

We focus on the *control data flow graph (CDFG)* as an *internal representation (IR)*. A CDFG consists of a set of control nodes N_{cfg} and control edges E_{cfg} . The *control nodes* are a set of basic blocks. Each control node holds a number of instructions or computations that execute atomically. The *control edges* model the control flow relationships between the control nodes. The control nodes and control edges form a directed graph $G_{cfg}(N_{cfg}, E_{cfg})$. Each control node contains a set of operations. The data flow relationships between the operations in a particular control node can be viewed as a sequential list of instructions I or a data flow graph $G_{dfg}(V_{dfg}, E_{dfg})$. The conversion from I to G_{dfg} , and vice-versa, is trivial.

As an IR, CDFGs offer many advantages. Many high-level programming languages (Fortran, C, C++) can be compiled into CDFGs with slight modifications to the front end of pre-existing compilers; a pass converting a typical high-level IR into control

flow graphs and subsequently CDFGs is possible with minimal modification. Furthermore, this allows us to use the back-end of existing compilers to generate code for a variety of processors¹. Additionally, data flow analysis techniques (e.g. reaching definitions, liveness, constant propagation, etc.) can be applied directly to CDFGs. This allows us to leverage the many well-developed transformations to improve the quality of our hardware description. Most importantly, we believe that the CDFG can be mapped to a variety of different micro-architectures, making it a good IR for a hardware compiler.

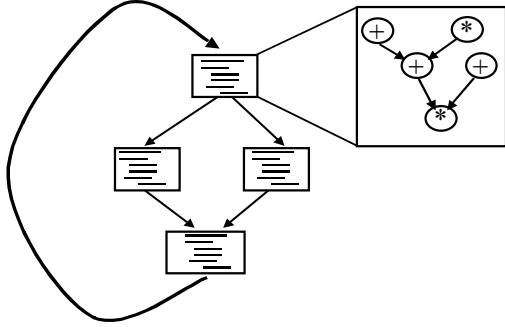


Figure 1: A Control Data Flow Graph

In this work, we examine the problem of mapping an application onto a micro-architecture. Specifically, we look at the problem of synthesizing a CDFG into some hardware description language (HDL). EDA tools – either academic or commercial – can perform optimization from that level on. This allows us to map the application onto many different styles of micro-architectures. We could map the application to a programmable architecture like an FPGA or even as a full-blown ASIC implementation. Now we briefly discuss methods of synthesizing data flow graphs, as these are the internal constructs to the CDFG.

2.2 Architecture Synthesis

The architecture synthesis problem has received much attention in the past 10 years. Scheduling and resource binding are the two major stages of architecture synthesis.

Given a set of operations with execution delays and a partial ordering, the scheduling problem determines the start time for each operation. Additional restrictions such as timing and resource constraints may be added to the problem, depending on the microarchitecture one is targeting. There are many algorithms for scheduling data flow graphs including Hu’s algorithm [3], list scheduling [4], and force-directed scheduling [5]. Hundreds of other heuristic scheduling algorithms targeting different objectives have been proposed.

Resource binding is the assignment of hardware resources to one or more operations; it is an explicit mapping between operations and resources. We refer an interested reader to DiMicheli’s book [4] for further details. There are many industrial and academic tools that are capable of synthesizing a data flow graph.

Our work assumes that there is a tool to synthesize data flow graphs to some structural hardware language. We look at the problem of synthesizing the “control” part of the CDFG. In the next section, we discuss our framework for CDFG synthesis.

3 Control Data Flow Graph Synthesis

3.1 Controlling the Path of Execution

Each control flow node consists of a set of inputs and a set of outputs. After the computations are completed, control is transferred to another control flow node. We must add a mechanism to direct the control flow i.e. a controller. We focus on two types of control – distributed control and centralized control.

Distributed control has several different entities that control the path of execution. Each control node has a local controller that determines the next control node in the execution sequence. Therefore, there are direct connections between control nodes. Every control node is equipped with an *execute* port that tells it when to begin execution. Additionally, each control node has a set of *control flow indicator (CFI)* ports. There is a CFI port for each of the different control nodes that may follow this node in execution. Equivalently, there is a CFI for each control edge emanating from a control node. A CFI port connects to the *execute* port of other control nodes. Figure 2 a) illustrates a simple example of distributed control.

Often, the control flow depends on the result of the computations local to the currently executing control node. For example, the condition for control flow may depend if a local variable is greater than zero. In this case, the control node must transmit the condition to the centralized controller. Then, the controller will determine the next control node in the execution sequence.

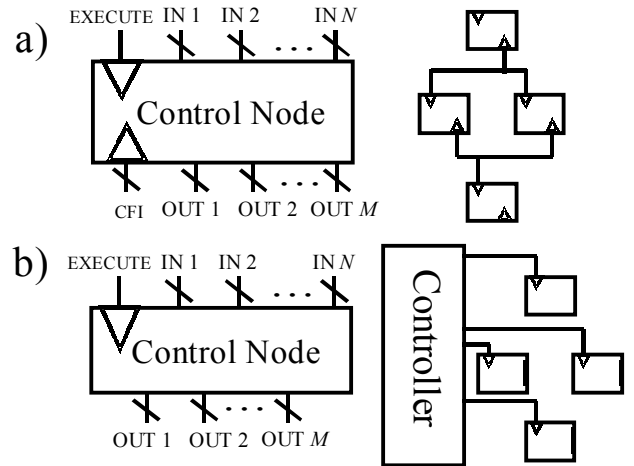


Figure 2: a) Distributed Control b) Centralized Control

Centralized control has one controller that determines the control node(s) that execute at any given instant. As with distributed control, each control node has an *execute* port that initiates the execution of the data flow graph embedded in the control node. Unlike distributed control, every *execute* port of control node is connected to the controller. Centralized control closely resembles the separation of control flow and data flow assumed by most

¹ Processors are common system-on-a-chip components.

high-level synthesis engines. Figure 2 b) gives an example of centralized control.

Of course, one could imagine many other control schemes. For example, there could be many distributed controllers, each of which controls a small number of nodes. A single centralized controller could control these distributed controllers. Hybrid local/global schemes are an interesting area of research but are out of the scope of this work.

3.2 Data Communication Schemes

In addition to determining the type of control for the CDFG, we must determine the method of data communication between the control nodes. Once again, there is a centralized and distributed method of data communication. A *centralized data communication scheme* passes the data through a centralized storage area such as a register bank or RAM block, depending on the amount of data. This allows a memory hierarchy scheme where data can be cached and large amount of data can be accessed by the CDFG.

A *distributed data communication scheme* passes the output data from the currently executing control node directly to the inputs of the control node(s) that might need the data later. The output of a control node may connect to multiple other control nodes including itself (in the case of loops).

As with most of engineering decisions, there are benefits and drawbacks to consider for each of the communication schemes. The distributed data communication scheme is simple to implement, as you do not have to worry about interfacing to bus and memory protocols. Additionally, the distributed scheme has direct connections, meaning that the communication between control nodes will occur quicker compared to the centralized scheme; data passing does not involve writing to and reading from a central memory back. Yet, the centralized scheme allows a sharing of resources. The distributed scheme will have more connections (interconnect), many of which will not be active at a particular time leading to a waste of communication resources. Furthermore, the increased connectivity between control nodes may have a negative impact on the circuit's area.

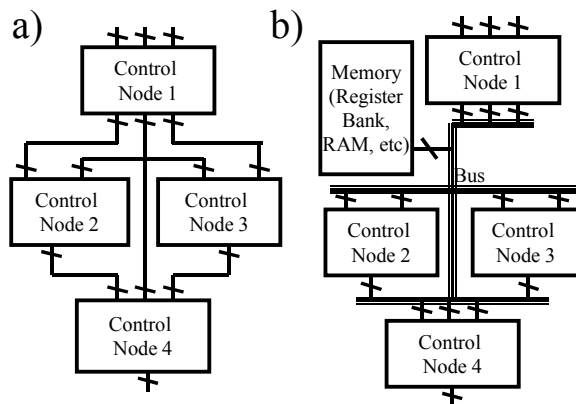


Figure 3: a) Distributed data communication b) Centralized data communication

In the next section, we describe how data flow analysis – more specifically SSA – is useful to reduce the amount of data exchange between control nodes.

4 Minimizing Inter-Node Communication

In order to determine the data exchange between the control nodes, we establish the relationship between where data is generated and where data is used for calculation. The specific place where data is generated is called its *definition point*. A specific place where data is used in computation is called a *use point*. The data generated at a particular definition point may be used in multiple places. Likewise, particular use point may correspond to a number of different definition points; the control flow dictates the actual definition point at any particular moment.

If data generated in one control node is used in a computation a second control node, these two control nodes must have a mechanism to transfer the data between them. A distributed data communication scheme has a direct connection between the two control nodes. If we used a centralized scheme, the first control node would transfer the data to memory and the second control node would access the memory for that data. Regardless of the scheme that we use, we should try to minimize the amount of inter-node communication. In a centralized scheme minimizing the inter-node communication would have a direct impact on the number of memory accesses. Also, it could reduce the number of components connected to the bus. By minimizing the data communication in the distributed scheme, we reduce the interconnect between the control nodes.

4.1 Static Single Assignment

We can determine the relationship between the use and definition points through static single assignment [6,7]. Static Single Assignment (SSA) renames variables with multiple definitions into distinct variables – one for each definition point.

We define a *name* to represent the contents of a storage location (e.g. register, memory). A name is unspecific to SSA. In non-SSA code, a name represents a storage location but we may not know the exact location; the precise location of the name depends on the control flow of the program. Therefore, we call a name in non-SSA code a *location*. SSA eliminates this confusion as each name represents a value that is generated at exactly one definition point. The SSA definition of a name is called a *value*.

In order to maintain proper program functionality, we must add Φ -nodes into the CDFG. Φ -nodes are needed when a particular use of a name is defined at multiple points. A Φ -node takes a set of possible names and outputs the correct one depending on the path of execution. Φ -nodes can be viewed as an operation of the control node. They can be implemented using a multiplexer. Figure 4 illustrates the conversion to SSA.

SSA is accomplished in two steps, first we add Φ -nodes and then we rename the variables at their definition and use points. There are several methods for determining the location of the Φ -nodes. The naïve algorithm would insert a Φ -node at each merging point for each original name used in the CDFG. A more intelligent algorithm – called the minimal algorithm – inserts a Φ -node at the iterated dominance frontier of each original name [6]. The semi-pruned algorithm builds smaller SSA form than the minimal algorithm. It calculates determines if a variable is local to a basic block and only inserts Φ -nodes for non-local variables [7]. The

pruned algorithm further reduces the number of Φ -nodes by only inserting Φ -nodes at the iterated dominance frontier of variables that are live at that time [8]. After the position of the Φ -nodes is determined, there is a pass where the variables are renamed.

The minimal method requires $O(|E_{cfg}| + |N_{cfg}|^2)$ time for the calculation of the iterated dominance frontier. The iterated dominance frontier and liveness analysis must be computed during the pruned algorithm. There are linear or near linear time liveness analysis algorithms [9,10,11]. Therefore, the pruned method has the same asymptotic runtime as the minimal method.

We should suppress any unnecessary data communication between control nodes. Now we explain how to minimize the inter-node communication.

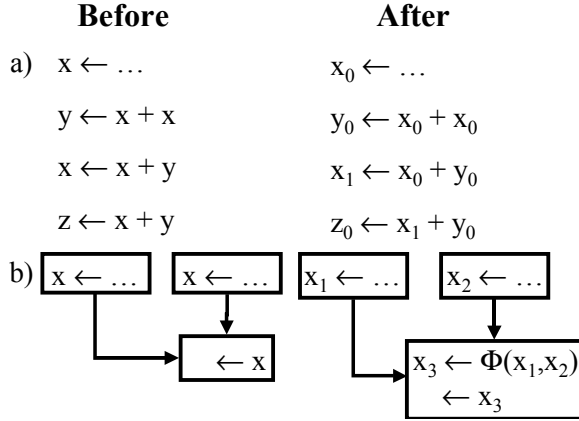


Figure 4: a) Conversion of Straight-line Code to SSA b) SSA Conversion with Control Flow

4.2 Minimizing Data Communication with SSA

SSA allows us to minimize the inter-node communication. The various algorithms used to create SSA all attempt to accurately model the actual need for data communication between the control nodes. For example, if we use the pruned algorithm for SSA, we eliminate false data communication by using liveness analysis, which eliminates passing data that will never be used again.

SSA allows us to minimize the data communication, but it introduces Φ -nodes to the graph. We must add a mechanism that handles the Φ -nodes. This can be accomplished by adding an operation that implements the functionality of a Φ -node. A multiplexer provides the needed functionality. The input names are the inputs to the multiplexer. An additional control line must be added for each multiplexer to determine that the correct input name is selected.

A fundamental limitation of using SSA in a hardware compiler is the use of the iterated dominance frontier for determining the positioning of the Φ -nodes. Typically, compilers use SSA for its property of a single definition point. We are using in another way – as a representation to minimize the data communication between hardware components (CFG nodes). In this case, the positioning of Φ -nodes at the iterated dominance frontier does not always optimize the data communication. We must consider spatial properties in addition to the temporal properties of the CDFG when determining the position of the Φ -nodes.

We illustrate our point with a simple example. Figure 5 b) exhibits traditional SSA² form for the code shown in Figure 5 a). The Φ -node is placed in control node 3. In the traditional SSA scheme, the data a_1 and a_2 generated in node 1 and 2 respectively is used in node 3, but only in the Φ -node. Then, the data a_3 is used in node 4. Therefore, there must be a communication connection from node 1 to node 3, node 2 to node 3 and node 3 to node 4 – a total of 3 communication links. In part c), the Φ -node is distributed to node 4. Then, we only need a communication connection from node 1 to node 4 and node 2 to node 4, a total of 2 communication links.

From this example, we can see that traditional Φ -node placement is not always optimal in terms of data communication. This arises because Φ -nodes are traditionally placed in a temporal manner. The iterated dominance frontier is the first place in the timeline of the program where the two (or more) locations of a variable merge. But, as you can see, this is not necessarily the only place where they can be placed. When considering hardware compilation, we must think spatially as well as temporally. By moving the position of the Φ -nodes, it is possible to achieve a better layout of our hardware design. In order to reduce the data communication, we must consider the number of uses of the value that a Φ -node defines as well as the number of values that the Φ -node takes as an input.

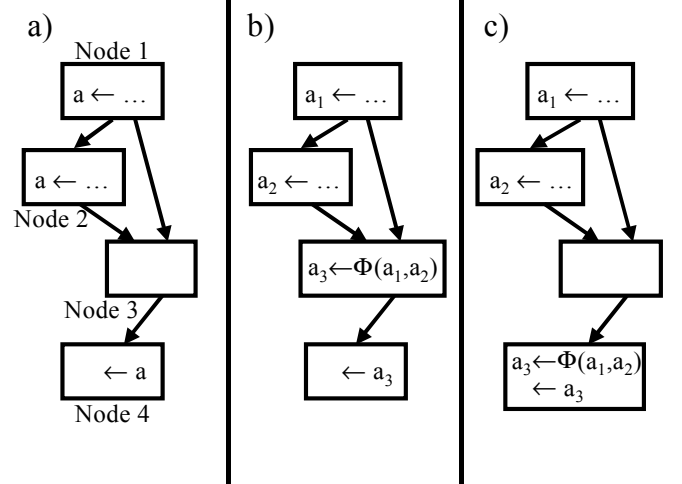


Figure 5: Simple example showing the benefit of distributing the Φ -node. Part a) shows a CFG before SSA. Part b) shows SSA form. Part c) displays SSA form with the Φ -node spatially distributed.

4.3 An Algorithm for Spatially Distributing Φ -nodes

The first step of spatially distributing Φ -nodes is determining which Φ -nodes should be moved. We assume that we are given the correct temporal positioning of the Φ -nodes according to some SSA algorithm (e.g. minimal, semi-pruned, pruned). The movement of a Φ -node depends on two factors. The first factor is

² We use the terms “traditional SSA” and “temporal SSA” interchangeably to mean the SSA introduced by Cytron et al. [6].

the number of values that the Φ -node must choose between. We call this the number of Φ -node *source values* s . The second factor is the number of uses that the value of the Φ -node defines. We call this the Φ -node *destination value* d . Taking Figure 5 as an example, the Φ -node source values are a_1 and a_2 whereas the Φ -node destination value is a_3 . Determining s is simple; we just need to count the number of source values in the Φ -node. Finding the number of uses of the destination value is a more difficult. We can use def-use chains [12], which can be calculated during SSA.

The relationship between the amount of communication links C_T needed for a Φ -node in temporal SSA and the number of communication links C_S in spatial SSA is:

$$C_T = s + d \quad C_S = s \cdot d$$

Using these relationships, we can easily determine if spatially moving a Φ -node will decrease the total amount of inter-node data communication. If C_S is less than C_T , then moving the Φ -node is beneficial. Otherwise, we should keep the Φ -node in its current location.

After we have decided on which Φ -nodes we should move, we must determine the control node(s) where we should move the Φ -node. This step is rather easy, as we move the Φ -node from its original location to control nodes that have a use of the definition value of that Φ -node. It is possible that by moving the Φ -node, we increase the total number of Φ -nodes in the design. But, we are decreasing the total amount of inter-node data communication. Therefore, the amount of data communication is not directly dependent on number of Φ -nodes.

1. Given a CDFG $G(N_{cfg}, E_{cfg})$
2. perform_SSA(G)
3. calculate_def_use_chains(G)
4. remove_back_edges(G)
5. topological_sort(G)
6. **for each** node $n \in N_{cfg}$
7. **for each** Φ -node $\Phi \in n$
8. $s \leftarrow |\Phi.sources|$
9. $d \leftarrow |\text{def_use_chain}(\Phi.dest)|$
10. **if** $s \cdot d < s + d$
11. move_to_spatial_locations(Φ)
12. restore_back_edges(G)

Figure 6: Spatial SSA Algorithm

It is possible that a use point of the definition value of Φ -node Φ_1 is another Φ -node Φ_2 . If we wish to move Φ_1 , we add the source values of Φ_1 into the source values of Φ_2 ; obviously, this action changes the number of source values of Φ_2 . In order to account for such changes in source values, we must consider moving the Φ -nodes in a topologically sorted manner based on the CDFG control edges. Of course, any back control edges must be removed in order to have valid topologically sorting. We can not move Φ -nodes across back edges as this can induce dependencies between the source value and the destination value of previous

iterations i.e. we can get a situation where $b_i \leftarrow \Phi(b_i, \dots)$. The source value b_i was produced in a previous iteration by that same Φ -node. The complete algorithm for spatially distributing Φ -node to minimize data communication is outlined in Figure 6.

Theorem 4.1: Given an initially correct placement of a Φ -node, the functionality of the program remains valid after moving the Φ -node to the basic block(s) of all the use point(s) of the Φ -node's destination value.

Proof: There are two cases to consider. The first case is when the use point is a normal computation. The second case is when a use point is Φ -node, itself.

We consider the former case first. When we move the Φ -node from its initial basic block, we move it to the basic blocks of every use point of the Φ -node's destination value d . Therefore, every use of the d can still choose from the same source values. Hence, if the Φ -node source values were initially correct, the use points of d remain the same after the movement. We must also insure that moving the Φ -node does not cause some other use point that uses the same name but has a different value. The Φ -node will not move past another Φ -node that has the same name because by construction of correct initial SSA, that Φ -node must have d as one of its source values.

The proof of the second case follows similar lines to that of the first one. The only difference is that instead of moving the initial Φ -node Φ_i to that basic block, we add the source values to the Φ -node Φ_u that uses d . If we move Φ_i before Φ_u , then the functionality of the program is correct by the same reasoning of the first part of proof. Assuming that the temporal SSA algorithm has only one Φ -node per basic block per name, we can add the source values of Φ_i to Φ_u while maintain the correct program functionality.

Theorem 4.2: Given a correct initial placement of Φ -nodes, the spatial SSA algorithm maintains the correct functionality of the program.

Proof: The algorithm considers the Φ -nodes in a topologically sorted manner. As a consequence of Theorem 4.1, the movement of a single Φ -node will not disturb the functionality of the program hence the Φ -node will not move past another value definition point with the same name. Since we are considering the Φ -nodes in forward topologically sorted order, the movement of any Φ -node will never move past a Φ -node which has yet to be considered for movement. Also, Φ -node can never move backwards across an edge (remember we remove back edges). Therefore, the algorithm will never move a value definition point past another value definition point with the same name. Hence every use preserves the same definition after the algorithm completes. This maintains the functionality of the program.

5 Experimental Results

To measure the effectiveness of using SSA to minimize data communication between control nodes, we examined a set of DSP functions (see Table 1). DSP functions typically exhibit a large amount of parallelism making them ideal for hardware. The DSP functions were taken from the MediaBench test suite [13]. The files were compiled into CDFGs using the SUIF compiler infrastructure [14] and the Machine-SUIF [15] backend.

Table 1: MediaBench functions

Application	C File	Description
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
rasta	fft.c	Fast Fourier Transform
rasta	noise_est.c	Noise estimation functions
Function	Name	# Control Nodes
adpcm_coder	adpcm1	33
adpcm_decoder	adpcm2	26
internal_expland	convolve1	101
internal_filter	convolve2	101
compress_output	jctrans	33
Decode MPEG2 Intra Block	getblk1	75
Decode MPEG2 Non Intra Block	getblk2	60
decode_motion_vector	motion	15
FAST	fft1	14
FR4TR	fft2	76
comp_noise power	noise_est1	153
Det	noise_est2	12

We performed SSA analysis with the SSA library built into Machine-SUIF. The library was initially developed at Rice [16] and recently adapted into the Machine-SUIF compiler.

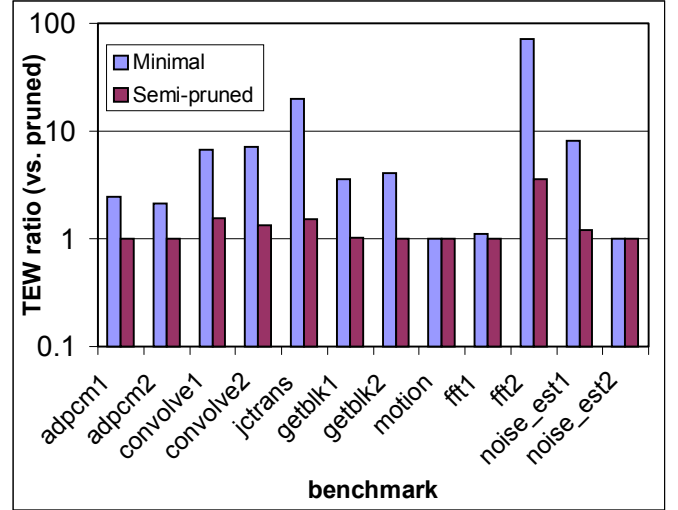
First, we compare the amount of data flow between the control nodes using the different SSA algorithms. Given two control nodes i and j , the *edge weight* $w(i, j)$ is the amount of data communicated (in bits) from control node i to control node j . The *total edge weight (TEW)* is:

$$TEW = \sum_i \sum_j w(i, j)$$

Figure 7 is a comparison of edge weights using three different algorithms for positioning the Φ -nodes. We compare the minimal, semi-pruned and pruned algorithms. Recall that the pruned algorithm is the best algorithm in terms of reducing the number of Φ -nodes, but worst in runtime. The minimal algorithm produces many Φ -nodes, but has small runtime. The semi-pruned algorithm provides a middle ground in terms of runtime and quality of result.

We divide the TEW of the minimal and semi-pruned algorithm (respectively) by the TEW of the pruned algorithm. We call this the *TEW ratio*. We use the pruned algorithm as a baseline

because it consistently produces the smallest TEW. Referring to Figure 7, the TEW of the minimal algorithm is much worse than that of the pruned algorithm. For example, in the benchmark `fft2`, the TEW of the minimal algorithm is over 70 times that of the TEW of the pruned algorithm. The semi-pruned algorithm yields a TEW that is smaller than that of the minimal algorithm, but still slightly larger than the TEW of the pruned algorithm. All algorithms have the same asymptotic runtime and the actual runtimes for all the algorithms over all the benchmarks were very small (under 1 second). Therefore, we feel that one should use the pruned algorithm as it minimizes data communication much better than the other two algorithms. Furthermore, the actual additional runtime needed to run the pruned algorithm is miniscule.

**Figure 7: Comparison of total edge weight (TEW) between the minimal and semi-pruned TEW and the pruned TEW**

Each of the algorithms we compared attempt to minimize the number of Φ -nodes, and not the data communication. There is obviously a relationship between the number of Φ -nodes and the amount of data communication. Every Φ -node defines additional data communication, but there can be inter-node data transfer without Φ -nodes. Furthermore, as we pointed out in Section 4.2, minimizing the number of Φ -nodes does not directly correspond to minimizing the data communication.

In Figure 8, we compare the ratio of Φ -nodes and the ratio of TEW using the minimal and pruned algorithms. As you can see, the number of Φ -nodes is highly related to the amount data communication. As the Φ -node ratio increases, the TEW ratio increases. Correspondingly, a large Φ -node ratio corresponds to a large TEW ratio. This lends validation to the using SSA algorithms to first minimize inter-node communication and then use the spatial Φ -node repositioning to further reduce the data communication. In other words, minimizing the number of Φ -nodes is a good objective function to initially minimize data communication.

Our next set of experiments focus on using spatial SSA Φ -node distribution to further minimize the amount of data communication. Figure 9 shows the number of Φ -nodes that are spatially distributed by the spatial SSA algorithm. We can see that these Φ -nodes are fairly common; in some of the benchmarks,

over 35% of the Φ -nodes are spatially moved. The average number of distributed Φ -nodes over all the benchmarks is 11.65%, 18.21% and 13.56%³ for the pruned, semi-pruned and minimal algorithms, respectively.

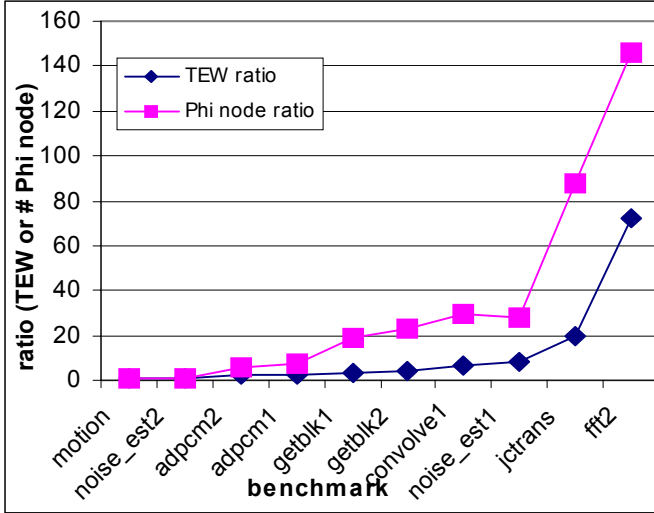


Figure 8: A comparison of total edge weight (TEW) and the number of Φ -nodes using the minimal and pruned algorithms.

Figure 10 gives the percentage of TEW improvement we achieve by spatially distributing the nodes. By spatially distributing the Φ -nodes, we reduce the TEW by 1.80%, 4.77% and 8.16% in the pruned, semi-pruned and minimal algorithms, respectively. We believe the small amount of improvement in TEW can be attributed to two things. First of all, the TEW contributed by the Φ -nodes is only a small portion of the total TEW. Also, when the number of Φ -nodes is small, the number of Φ -nodes to distribute is also small. This is apparent in the increasing trend seen by the pruned, semi-pruned and minimal algorithms. There are many Φ -nodes when we use the minimal algorithm and correspondingly, there TEW improvement of the minimal algorithm is the 8.16%. Conversely, the number of Φ -nodes in the pruned algorithm is small and the TEW improvement is also small.

6 Related Work

The idea of hardware compilation has been discussed since the 80's. At that time, it was under the guise of silicon compilation and related closely to what is referred to as behavioral synthesis nowadays.

The past 15 years have brought about a number of platforms that take high-level code and generate a hardware configuration for that platform. The PRISM project [17] took functions implemented in a subset of C and compiled them to their FPGA-like architecture. The Garp compiler [18] takes automatically maps C code to their MIPS + FPGA architecture. The DeepC compiler [19] is the most similar to our work, as it synthesizable Verilog from C or Fortran. These are some of the more prevalent

academic works in hardware compilation. The SystemC [1] and SpecC [2] languages have created much industrial interest in hardware compilation. Many companies including Synopsis and Cadence are exploring hardware compilation from these two languages.

Many compiler techniques use SSA for analysis or transformation [20,21,22]. Also, there have been modifications of SSA form [23,24]. To the best of our knowledge, this is the first work that considers SSA form for hardware compilation.

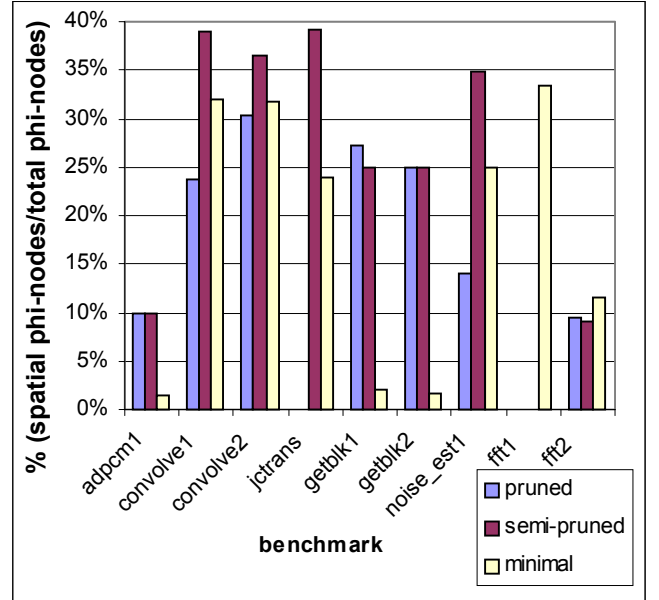


Figure 9: Comparison of the number of spatially distributed Φ -nodes and the total number Φ -nodes using the three SSA algorithms.

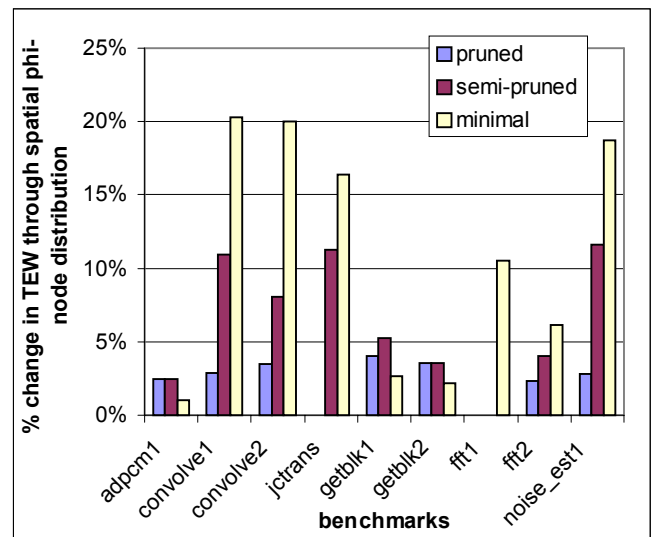


Figure 10: The percentage change in of total edge weight when we distribute the Φ -nodes using the three SSA algorithms.

³ Not all of the benchmarks are included in Figure 9; the omitted benchmarks have 0 Φ -nodes that should be distributed, but these benchmarks are included in the averages.

7 Conclusion

In this work, we presented methods needed for hardware compilation. First, we described a framework for compiling a high-level application to an HDL. The framework includes methods for transforming a traditional compiler IR to an RTL-level HDL. We illustrated how to transform the IR into a CDFG form. Using the CDFG form, we explained methods to control the path of execution. Furthermore, we gave methods for communicating data between the control nodes of the CDFG.

We examined the use of SSA to minimize the amount of data communication between control nodes. We showed that through the use of SSA, we reduced the amount of inter-node communication by 70 times in one DSP function. We showed a shortcoming of SSA when it is applied to minimizing data communication. The temporal positioning of the Φ -node is not optimal in terms of data communication. We formulated an algorithm to spatially distribute the Φ -node to minimize the amount of data communication. We showed that this spatial distribution can decrease the data communication by 20% for some DSP functions.

As future work, we plan to integrate this functionality into the hardware compiler we are creating for reconfigurable systems. Then, we can examine the actual effect that minimizing data communication has on hardware parameters such as area, power, delay, etc. Additionally, we hope to find a distribution of Φ -nodes such that the data communication is provably minimal.

References

- [1] Open SystemC Initiative, <http://www.systemc.org>.
- [2] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauser, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, 2000.
- [3] T. C. Hu, "Parallel Sequencing and Assembly Line Problems", *Operations Research*, no. 9, pp. 841-848, 1961.
- [4] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., New York, 1994.
- [5] P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on CAD/ICAS*, vol. 8, no. 6, pp. 661-679, July 1989.
- [6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadek, "An Efficient Method of Computing Static Single Assignment", *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1989.
- [7] P. Briggs, K. Cooper, T. Harvey and L. Simpson, "Practical Improvements to the Construction and Destruction of Static Single Assignment Form", *Software Practice and Experience*, vol. 28, no. 8, pp. 859-881, July 1998.
- [8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadek, "Efficiently Computing Φ -nodes On-the-Fly", *ACM Transactions on Programming Languages and Systems*, October 1991.
- [9] S. L. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis", *Journal of the ACM*, vol. 23, no. 1, pp. 172-202, January 1976.
- [10] J. B. Karn and J. D. Ullman, "Global Data Flow Analysis and Iterative Algorithms", *Journal of the ACM*, vol. 23, no. 1, pp. 158-171, January 1976.
- [11] K. Kennedy, "A Survey of Data Flow Analysis Techniques", *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.
- [12] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, 1997.
- [13] C. Lee, M. Potkonjak and W. H. Maggione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 1997.
- [14] 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.
- [15] M. D. Smith and G. Holloway, *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*, Division of Engineering and Applied Sciences, Harvard University, <http://www.eecs.harvard.edu/machsuiif/>.
- [16] P. Briggs, T. Harvey and L. Simpson, *Static Single Assignment Construction*, Implementation documentation, 1996. Available at <ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps>.
- [17] A. Smith, M. Wazlowski, L. Agarwal, T. Lee, E. Lam, P. Athans, H. Silverman and S. Ghosh, "PRISM II Compiler and Architecture", *Proceedings of IEEE Workshop on FPGA-based Custom Computing Machines*, April, 1993.
- [18] T. J. Callahan, J. R. Hauser and J. Wawrzynek, "The Garp Architecture and C Compiler", *IEEE Computer*, vol. 33, no. 4, April, 2000.
- [19] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua and S. Amarasinghe, "Parallelizing Applications into Silicon", *Proceedings of Field-Programmable Custom Computing Machines*, 1999.
- [20] B. Alpern, M. N. Wegman and F. K. Zadek, "Detecting Equality of Variables in Programs", *Proceedings of Principals of Programming Languages*, Jan. 1988.
- [21] P. Briggs and K. D. Cooper, "Effective Partial Redundancy Elimination", *Proceedings of Programming Language Design and Implementation*, June 1994.
- [22] P. Briggs, K. D. Cooper and L. T. Simpson, "Value Numbering", *Software – Practice and Experience*, vol. 27, no. 6, June 1997.
- [23] L. Carter, B. Simon, B. Calder, L. Carter and J. Ferrante, "Predicated Static Single Assignment", *Proceedings of Parallel Architectures and Compilation Techniques*, Oct. 1999.
- [24] W. Amme, N. Dalton, J. von Ronne and M. Franz, "SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form", *Proceedings of Programming Language Design and Implementation*, June 2001.

