High-Level Data Communication Optimization For Reconfigurable Systems

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

Ryan Kastner
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106
kastner@ece.ucsb.edu

ABSTRACT
This paper describes methods for synthesizing the internal representation of a compiler into a hardware description language in order to program reconfigurable hardware devices. We demonstrate the usefulness of static single assignment (SSA) in reducing the amount of data communication in the hardware. However, the placement of Φ-nodes by current SSA algorithms is not optimal in terms of minimizing data communication. We propose an improved SSA algorithm which optimally places Φ-nodes, further decreasing area and communication latency. Our algorithm reduces the data communication (measured as total edge weight in a control data flow graph) by as much as 20% for some applications as compared to the best-known SSA algorithm – the pruned algorithm. We also show that our algorithm frequently leads to increased overall area, and describe future modifications to our model that should correct this shortcoming.

1. INTRODUCTION
Reconfigurable devices contain logic that can quickly be reprogrammed as often as needed. This has led to a revolution in the way designers conceptualize hardware systems, as the very logic that drives circuitry can be customized as often as needed. Reconfigurable hardware is usually realized via Field Programmable Gate Array (FPGA) technology. Increasingly, this hardware is being incorporated into computing systems, often coupled with one or more microprocessor or ASIC devices on the same chip. The reconfigurable components of the system provide fast, flexible logic at a very low cost. These components can be modified and re-implemented, much like software programs. Therefore, it has become attractive to design hardware algorithms in a high-level programming language, and compile this code into actual hardware logic (rather than software binary form).

The compiler straddles the boundary between application and hardware, making it a natural area to perform reconfigurable system exploration. The compiler can already map portions of the application to different processors by simply emitting code. In order to complete the system exploration space – one with processors, ASIC and reconfigurable components, we need a path from the compiler to a hardware description language (HDL). This HDL can then be synthesized into reconfigurable circuitry.

An area of extreme importance is the translation of the compiler's intermediate representation (IR) to a form that is suitable for synthesis to hardware. During this translation, we should attempt to exploit the existing concurrency of the application and discover additional parallelism. Also, we should determine the types of hardware specialization that will increase the efficiency of the application. Finally, we must take into account the hardware properties of the circuit, e.g. power dissipation, critical path and interconnect area.

Static single assignment [1,2] transforms the IR such that each variable is defined exactly once. It is an ideal transformation for hardware because side effects of the transformation, Φ-nodes, are easily implemented in hardware as multiplexors. Furthermore, it creates a one-to-one mapping between each variable and its corresponding value, which allows the compiler to identify each individual signal uniquely. It has been used in many projects where the final output is an HDL [3,4,5]. Yet, SSA was originally developed to enable optimizations for microprocessor architectures; it was not originally meant for hardware synthesis.

In this paper, we describe SSA and its effect on the optimization of hardware properties of the circuit. We show how SSA can be used to minimize data communication; this has a direct effect on the area, amount of interconnect and delay of the final circuit. Furthermore, we show that SSA in its original form is not optimal in terms of data communication and give an optimal algorithm for the placement of Φ-nodes to minimize the amount of data communication.

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

2. PRELIMINARIES

2.1 Control Data Flow Graphs
We focus on the control data flow graph (CDFG) as a *model of computation (MOC)* for the internal representation (IR) of the compiler. The CDFG offers several advantages over other models of computation. Most compilers have an IR that can easily be transformed into a CDFG. Therefore, this allows us to use the back-end of a compiler to generate code for a variety of processors. Furthermore, the techniques of data flow analysis (e.g. reaching definitions, live variables, constant propagation, etc.) can be applied directly to CDFGs. Finally, many high-level programming languages (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. Most

importantly, we believe that the CDFG can be mapped to a variety of different microarchitectures[1]. All of these reasons indicate that the CDFG is a good MOC for investigating the performance of mapping different parts of the application across a wide variety of SOC components.

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. A pictorial view of a CDFG is represented in Figure 1.

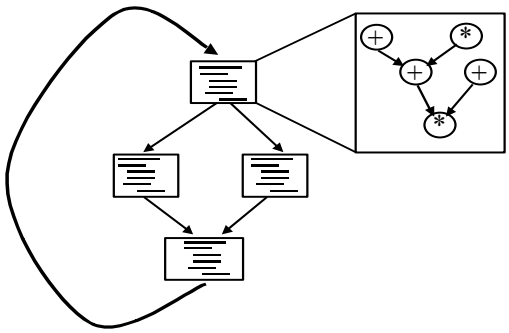

Figure 1: A Control Data Flow Graph

In this work, we examine the problem of manipulating a CDFG such that the resulting hardware exhibits enhanced performance. Our work assumes that there is a tool to synthesize a CDFG into some hardware description language (HDL). (EDA tools – either academic or commercial – can perform optimization from that level on.) We have built such a tool for system synthesis, and have used it to obtain the results of this work. We refer the interested reader to [6] for more details.

3. MINIMIZING INTER-NODE COMMUNICATION

In order to determine the data exchange between nodes in a CDFG, 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, a 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 in a second control node, these two control nodes must have a mechanism to transfer the data between them. One method of communicating data between two control nodes is to send it across a direct connection between the two control nodes. This requires a set of wires to exist between the control nodes. In an alternate scheme, the first control node transfers the data to memory and the second control node reads the memory to access the data. Therefore, with the latter communication method minimizing the inter-node communication would have a direct impact on the number of memory accesses, whereas with the former method the interconnect between the control nodes would be reduced. However, in both scenarios real performance boosts can be realized through communication optimization. Thus, regardless of the data communication method used, we should try to generically model and minimize inter-node communication.

3.1 Static Single Assignment

We can determine the relationship between the use and definition points through static single assignment [1,2]. 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 multiplexor. Figure 2 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 (IDF) of each original name [1]. The semi-pruned algorithm builds smaller SSA form than the minimal algorithm. It determines if a variable is local to a basic block and only inserts Φ-nodes for non-local variables [2]. The pruned algorithm further reduces the number of Φ-nodes by only inserting Φ-nodes at the IDF of variables that are live at that time [7]. 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 [8]. 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.

[1] We refer to a microarchitecture as a register transfer level description.

3.2 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 multiplexor provides the needed functionality. The input names are the inputs to the multiplexor. An additional control line must be added for each multiplexor to determine that the correct input name is selected.

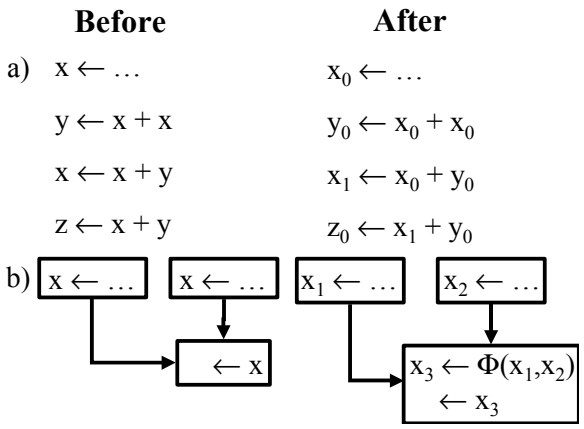

Figure 2: a) Conversion of Straight-line Code to SSA b) SSA Conversion with Control Flow

A fundamental limitation of using SSA in a hardware compiler is the use of the IDF for determining the positioning of the Φ-nodes. Typically, compilers use SSA for its property of a single definition point. We are using it 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 define *temporal placement* as the traditional placement of a Φ-node at the IDF, and introduce *spatial placement* as the placement of a Φ-node at its use point(s).

We illustrate this concept with a simple example. Figure 3a exhibits traditional SSA[2] form as well as the corresponding floorplan, containing control nodes a through e. The Φ-node is placed in control node d. In the traditional SSA scheme, the data values x_2, x_3, and x_4 (from nodes a, b, and c) are used in node d, but only in the Φ-node. Then, the data x_5 is used in node e. Therefore, there must be a communication connection from node a to node d, node b to node d and node c to node d, as well as a connection from node d to node e – a total of 4 communication links. In Figure 3b, the Φ-node is spatially distributed to node e. Then, we only need a communication connection from nodes a,b, and c to node e, a total of 3 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. Clearly, however, 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.

3.3 An Algorithm for 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). At this point, we have no knowledge of the actual cost of communication between any two basic blocks (as this will be determined later during layout). Thus, we choose to consider the communication cost between any two blocks as a unit cost of 1. (At a later design step, an annotated CDFG could return to this phase with complete cost information. In this case, a more refined model should be used.) The movement of a Φ-node depends on two factors. The first factor is 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 3a as an example, the Φ-node source values are x_2, x_3, and x_4 whereas the Φ-node destination value is x_5. 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 more difficult. We can use def-use chains [9], which can be calculated during SSA.

The relationship between the number 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:

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

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

[2] We use the terms "traditional SSA" and "temporal SSA" interchangeably to mean the SSA introduced by Cytron et al. [1].

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.

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 topological 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 $b_1 \leftarrow \Phi(b_1, ...)$. The source value b_1 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 4.

```
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 ∈ N_cfg
7.     for each Φ-node Φ ∈ n
8.        s ← |Φ.sources|
9.        d ← |def_use_chain(Φ.dest)|
10.          if s · d < s + d
11.             move_to_spatial_locations(Φ)
12. restore_back_edges(G)
```

Figure 4: Spatial SSA Algorithm

Theorem 3.1: 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 ensure 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

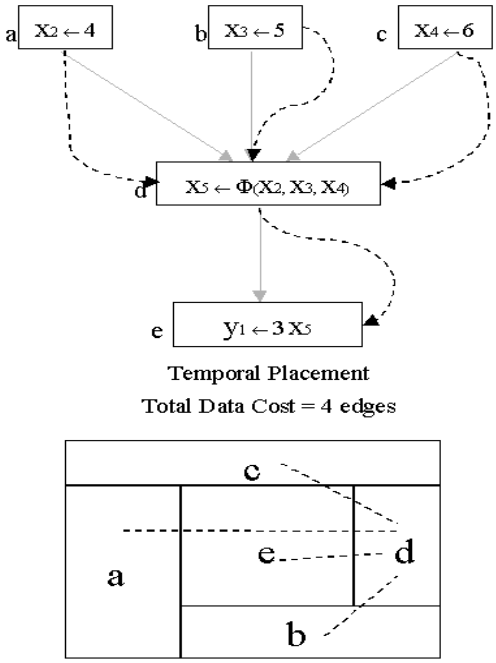

Figure 3a: SSA form and the corresponding floorplan (dotted edges represent data communication, and grey edges represent control). Data communication = 4 units.

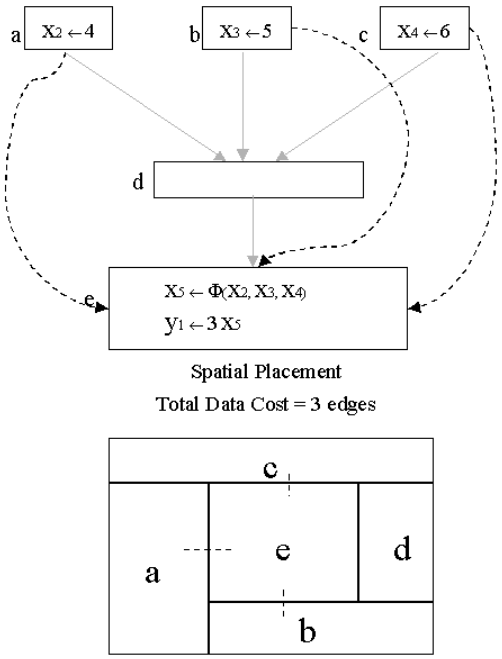

Figure 3b: SSA form with the Φ-node spatially distributed, as well as the corresponding floorplan. Data communication = 3 units.

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 maintaining the correct program functionality. □

Theorem 3.2: 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 3.1, 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 (recall that 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. □

Theorem 3.3: Given a floorplan where all wire lengths are unit length, the Spatial SSA Algorithm provides minimal data communication.

Proof: The source values of any given phi function are individual control nodes, and the cardinality of these nodes shall be referred to as *s*. Likewise, the destination points of any phi function are individual control nodes, and their cardinality will be referred to as *d*. The number of control nodes which define a given phi function (i.e. the number of phi nodes for a given phi function) will be referred to as *n*. The amount of data communication that this algorithm can reduce is restricted to the number of data edges coming into each phi node and the number of data edges coming out of each phi node. (The other data communication is already minimized, since SSA variables are actual data values. Therefore, SSA variables passed between control blocks are actual pieces of data that must be moved.) If a phi node is coalesced with its use point, then the number of out degree edges specifically leaving the phi node can be considered equal to zero. (The phi node's out degree data edges are now equal to the out degree of the use point, which cannot be reduced any further by the placement or removal of the phi node. Therefore the phi node's out degree of data will be considered equal to zero in this case.)

The total number of data communication points entering and exiting the phi nodes of a given phi function can be represented by a cost equation:

$$C = \sum_{\text{n } \Phi \text{ nodes}} (in + out)$$

where *in* is the number of inbound edges to each phi node and *out* is the number of outbound edges from each phi node.

In a floorplan where each edge has unit cost, this equation represents the total cost of this phi function in the graph.

In order to maintain correctness in a CDFG, every source value of a phi function must be coming into all phi nodes defining this function. (This is the only data that needs to enter a phi node.) Therefore, for all minimal cost cases, we can say that $in = s$ for every phi node and the data communication cost of the phi function can be restated as

$$C = ns + \sum_{\text{n } \Phi \text{ nodes}} out$$

since *s* is constant.

This leaves us with two values we can minimize: *n* (the number of total nodes defining a given phi function) and *out* (the out degree of a phi node), since *s* cannot be reduced (for correctness's sake). The most minimal cost we can have is when $n = 1$ or $out = 0$.

($n >= 1$, because at least one node must define the phi function. $out = 0$ is possible, as stated earlier.)

In the case that $out = 0$, the phi function will be coalesced with every use point of that function. That means that the total number *n* of nodes defining this function will equal *d* (the number of use points of the phi function). Therefore,

$$C = ns + \sum_{\text{n } \Phi \text{ nodes}} out = ns = d \cdot s = \mathbf{s} \cdot \mathbf{d}$$

(corresponding to spatial placement)

In the case that $n = 1$, that means that there is only one node defining a given phi function. This means that either a) there is a directed edge from this node to every use point or b) there is only one use point and this node has been coalesced with it.

In the case of part a, the total number of directed edges leaving the one phi node is equal to *d* (the number of use points) therefore

$$C = 1 * s + \sum_{\text{n } \Phi \text{ nodes}} out = s + out = \mathbf{s} + \mathbf{d}$$

(corresponding to temporal placement)

Part b is a special case of $C = s * d$ ($n = 1$, $out = 0$).

Therefore, we can minimize the total in/out degree of the phi node(s) by minimizing the equations ($C = s + d$, $C = s * d$). This corresponds to either choosing temporal placement (in the case of $s + d < s * d$) or choosing spatial placement (if $s + d > s * d$). This minimization of the degree of the phi node(s) leads to minimal data communication in a CDFG with edges having unit communication cost. □

4. EXPERIMENTAL RESULTS

To measure the effectiveness of using SSA to minimize data communication between control nodes, we examined a set of DSP functions. DSP functions typically exhibit a large amount of parallelism making them ideal for hardware. The DSP functions were taken from the MediaBench test suite [10]. The files were compiled into CDFGs using the SUIF compiler infrastructure [11] and the Machine-SUIF [12] backend. Then, each of the benchmarks was synthesized using the Synopsys Behavioral

Compiler for architectural synthesis followed by the Synopsys Design Compiler for logic synthesis.

We performed SSA analysis with the SSA library built into Machine-SUIF. The library was initially developed at Rice [13] and recently integrated 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 5 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 5, 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.

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 3.2, minimizing the number of Φ-nodes does not directly correspond to minimizing the data communication.

In Figure 6, we compare the ratio of Φ-nodes and the ratio of TEW using the minimal and pruned algorithms. Evidently, the number of Φ-nodes is highly related to the amount of 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 using SSA algorithms to first minimize inter-node communication and then using 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 7 shows the number of Φ-nodes that are spatially distributed by the spatial SSA algorithm. We can see

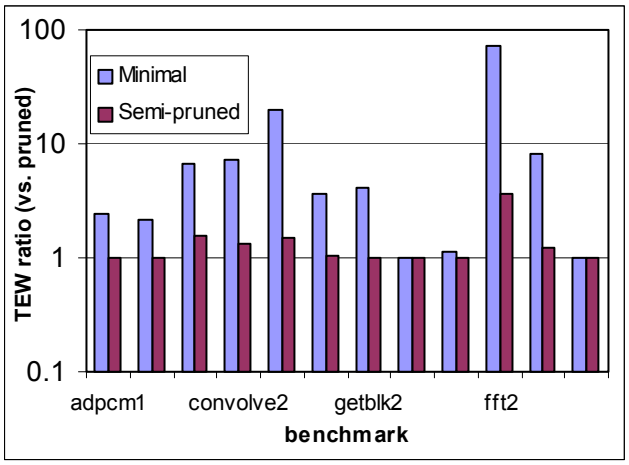

Figure 5: Comparison of total edge weight (TEW) between the minimal and semi-pruned TEW and the pruned TEW

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 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.

We ran the spatial algorithm through our system framework to determine the actual area improvements achieved by performing the Spatial SSA Algorithm to distribute the phi-nodes. The results are shown in Figure 9. The results are mixed and mostly negative. The chart plots the total area of the temporal (original) phi node placement divided by the total area of our algorithm's phi node placement. A result above 1 denotes that the temporal area is larger than the spatial area, meaning that our spatial phi node placement algorithm is beneficial. The benchmarks getblk1 and getblk2 benefit immensely from the spatial phi node placement.

[3] Not all of the benchmarks are included in Figures 7 and 8; the omitted benchmarks have 0 Φ-nodes that should be distributed, but these benchmarks are included in the averages.

The other benchmarks either exhibit higher total area due to spatial placement or the total area is approximately the same (i.e. the total area ratio is approximately equal to 1).

We believe that the results are somewhat negative for two reasons. First, as stated previously, the TEW reduction when using the spatial algorithm is not that large. The TEW reduction was 1.80%, 4.77% and 8.16% using the pruned, semi-pruned and minimal algorithms. Second, and more importantly, we have assumed that all wires are of unit length, which is a naïve estimation of circuit characteristics. Thus, the TEW is a flawed model, as it does not take into account the actual cost of communication between control nodes. (In Section 6, we conclude with future work intended to enhance this model.)

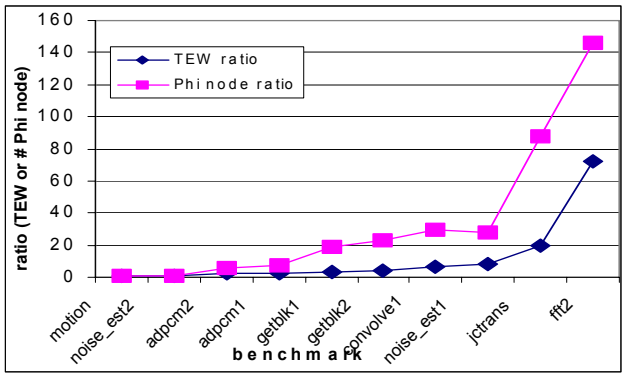

Figure 6: A comparison of total edge weight (TEW) and the number of Φ-nodes using the minimal and pruned algorithms.

5. RELATED WORK

The idea of hardware compilation has been discussed since the 1980s. At that time, it was researched 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 [14] took functions implemented in a subset of C and compiled them to their FPGA-like architecture. The Garp compiler [4] automatically maps C code to their MIPS + FPGA architecture. The DeepC compiler [15] is the most similar to our work, as it synthesizes Verilog from C or Fortran. These are some of the more prevalent academic works in hardware compilation. The SystemC [16] and SpecC [17] 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 [18,19,20]. To the best of our knowledge, this is the first work that considers SSA form for hardware compilation.

6. 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 provided 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 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 (measured as TEW) by 20% for some DSP functions. Additionally, we proved that if all data communication wire-lengths are of unit cost, the Spatial SSA Algorithm provides minimal data communication.

In practice, we found that our algorithm frequently increases total area of the circuit, which is a negative result. As future work, we plan to use a feedback mechanism from the hardware floorplanner to the compiler to incrementally derive more optimal results. This will enable us to annotate the CDFG with better wire length estimates (obtained during placement), rather than using the naïve unit wire length approximation assumed in this work. Presently, we only consider temporal and spatial placement of Φ- nodes (at the IDF or at use points), but frequently there is an intermediate range of possible placements between these two locations. We intend to explore the possibilities for Φ- node distribution across this range, while using more realistic wire lengths for higher accuracy. Additionally, we plan to account for the size of duplicated multiplexors. Placement of Φ- nodes will become an algorithmically harder problem, but will yield higher performance through further reduced data communication.

7. REFERENCES

[1] 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.

[2] 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.

[3] E. Waingold et al, "Baring it all to Software: The Raw Machine," *IEEE Computer*, Sep 1997.

[4] T. J. Callahan, J. R. Hauser and J. Wawrzynek, "The Garp Architecture and C Compiler", *IEEE Computer*, vol. 33, no. 4, April, 2000.

[5] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain and J. Granacki, "DEFACTO: A Design Environment for Adaptive Computing Technology", *Proceedings of the 6th Reconfigurable Architectures Workshop*, Springer-Verlag, 1999.

[6] R. Kastner, *Synthesis Techniques and Optimizations for Reconfigurable Systems*, PhD thesis, University of California, Los Angeles, 2002.

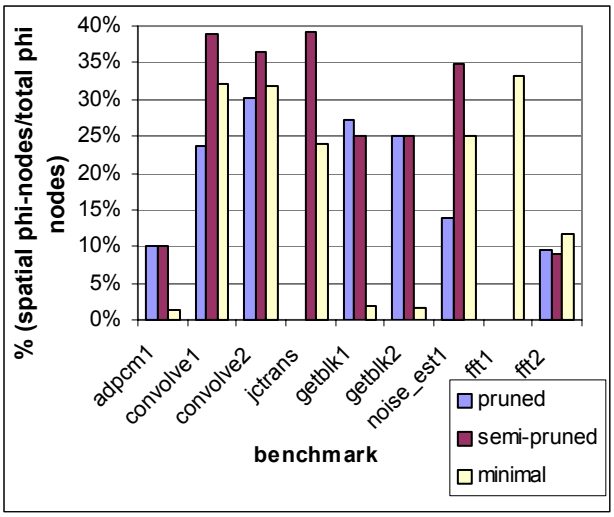

Figure 7: Comparison of the number of spatially distributed Φ-nodes and the total number of Φ-nodes using the three SSA algorithms.

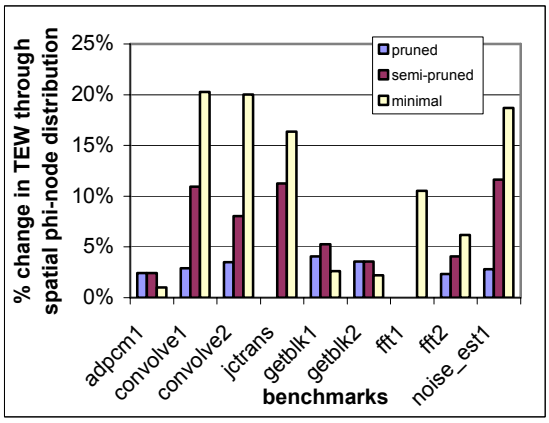

Figure 8: The percentage change in total edge weight when we distribute the Φ-nodes using the three SSA algorithms.

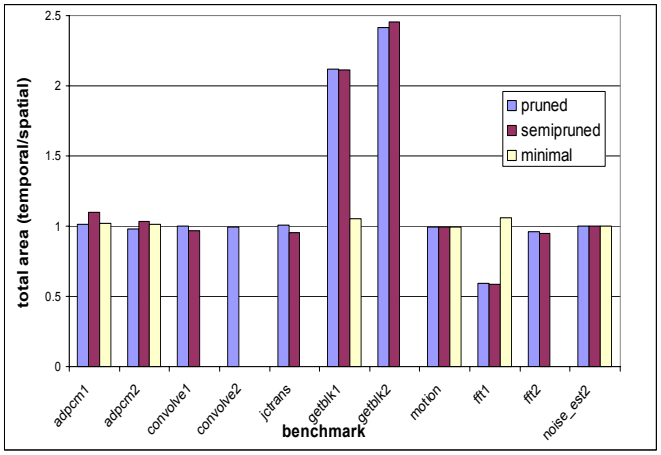

Figure 9: Comparison of the total area of the temporal versus spatial phi node placement for the three SSA algorithms. (Omitted benchmarks too large to synthesize.)

[7] 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.

[8] K. Kennedy. "A Survey of Data Flow Analysis Techniques", *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, 1997.

[10] 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.

[11] 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.

[12] 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/machsuif/

[13] 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.

[14] 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.

[15] 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.

[16] Open SystemC Initiative, http://www.systemc.org.

[17] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauser, S. Zhoa, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, 2000.

[18] B. Alpern, M. N. Wegman and F. K. Zadeck, "Detecting Equality of Variables in Programs", *Proceedings of Principals of Programming Languages*, Jan. 1988.

[19] P. Briggs and K. D. Cooper, "Effective Partial Redundancy Elimination", *Proceedings of Programming Language Design and Implementation,* June 1994.

[20] P. Briggs, K. D. Cooper and L. T. Simpson, "Value Numbering", *Software – Practice and Experience*, vol. 27, no. 6, June 1997.