Data Communication Estimation and Reduction for Reconfigurable Systems

Adam Kaplan Philip Brisk
Computer Science Department
University of California, Los Angeles
{kaplan,philip}@cs.ucla.edu

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

ABSTRACT

Widespread adoption of reconfigurable devices requires system level synthesis techniques to take an application written in a high level language and map it to the reconfigurable device. 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 a new 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 describe future modifications to our model that should increase the effectiveness of our methods.

1. Introduction

The advent of reconfigurable devices 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 design 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 – including 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, such as power dissipation, critical path and interconnect area.

Static single assignment (SSA) [1,2] transforms the IR such that each variable is defined exactly once. It is an ideal transformation for hardware because the side effects of the transformation, Φ-nodes, are easily implemented in hardware as multiplexers. Furthermore, SSA creates a one-to-one mapping between each variable and its corresponding value, which allows the compiler to identify each individual signal uniquely. In many ongoing projects, a compiler translates high-level algorithmic code to an HDL [3,4,5]. SSA could easily be incorporated into many of these projects as a compiler pass. 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 a 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

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. This justifies our selection of the CDFG as an 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 $G_{dfg}(V_{dfg}, E_{dfg})$.

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

3. Minimizing Inter-Node Communication

In order to determine the data exchange between nodes in a CDFG, we establish the relationship between the nodes in which data is generated and the nodes 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. A distributed data communication scheme has a direct connection between the two control nodes (i.e. one node controls the other's execution through a signal). If a centralized data communication scheme were used, the first control node would transfer the data to memory and the second control node would access the memory for that data. Therefore, in a centralized scheme minimizing the inter-node communication would have a direct impact on the number of memory accesses, and in a distributed scheme the interconnect between the control nodes would be reduced. However, in both schemes real performance boosts can be realized through communication optimization. Thus, regardless of the scheme that we use, 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 multiplexer. Figure 1 illustrates the conversion to SSA.

	Before	**After**
a)	$x \leftarrow \ldots$	$x_0 \leftarrow \ldots$
	$y \leftarrow x + x$	$y_0 \leftarrow x_0 + x_0$
	$x \leftarrow x + y$	$x_1 \leftarrow x_0 + y_0$
	$z \leftarrow x + y$	$z_0 \leftarrow x_1 + y_0$

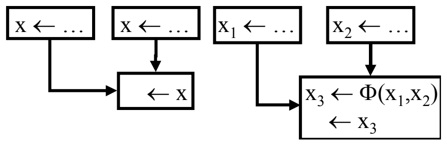
b)


Figure 1: a) Conversion of Straight-line Code to SSA b) SSA Conversion with Control Flow

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 IDF of a set of nodes is equivalent to the temporally nearest node at which their control paths rejoin. 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.

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, at the cost of introducing Φ-nodes to the graph. We must add a mechanism 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 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 illustrate our point with a simple example. Figure 2a exhibits traditional SSA[1] 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 2b, the Φ-node is 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.

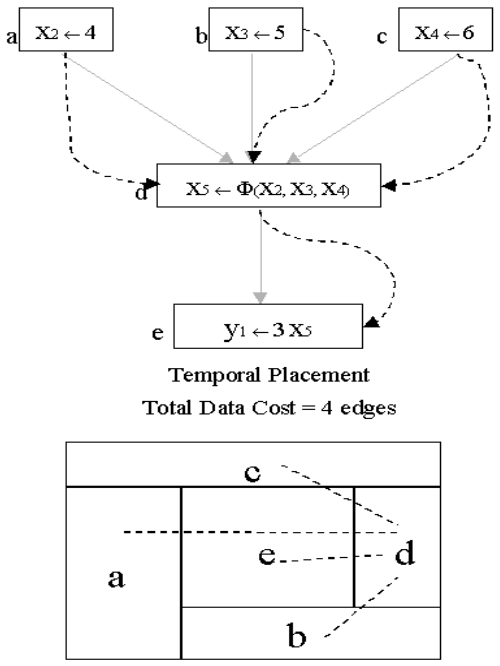


Figure 2a: SSA form and the corresponding floorplan (dotted edges represent data communication, and grey edges represent control). Data communication = 4 units.

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 IDF 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. The IDF is an excellent location to place Φ-nodes if your goal is shortening the liveness range of a variable. 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, in addition to 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). 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 2a 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.

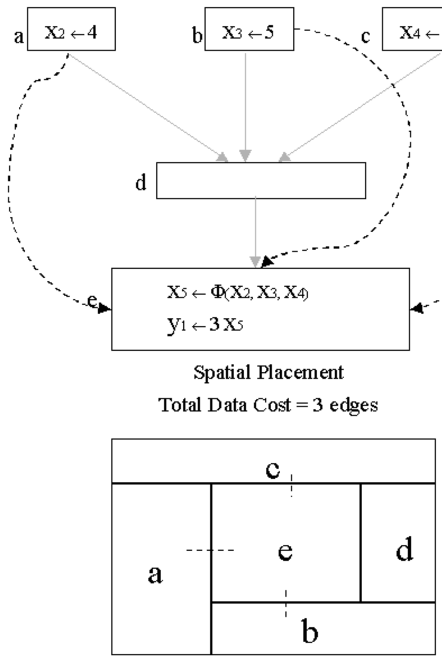


Figure 2b: SSA form with the Φ-node spatially distributed, as well as the corresponding floorplan. Data communication = 3 units.

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 at the IDF.

After we have identified a set of Φ-nodes to be moved, we must determine the control node(s) to which we should move each 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

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

Φ-node. It is possible that by moving the Φ-node, we increase the total number of Φ-nodes in the design. However, 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 $b_1 \leftarrow \Phi(b_1, \ldots)$. 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 3.

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.

Theorem 3.2: Given a correct initial placement of Φ-nodes, the spatial SSA algorithm maintains the correct 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.

The proofs of the proceeding theorems are removed for brevity. Please see [6] for further details.

```
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.        $d \leftarrow |$def_use_chain($\Phi.dest$)$|$
10.          if $s \cdot d < s + d$
11.             move_to_spatial_locations(Φ)
12. restore_back_edges($G$)
```

Figure 3: Spatial SSA Algorithm

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 candidates for hardware compilation. 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 given to our framework [6], which transforms the IR into synthesizable VHDL. The VHDL was then 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.

Our SSA algorithm is described as follows. 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 4 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.

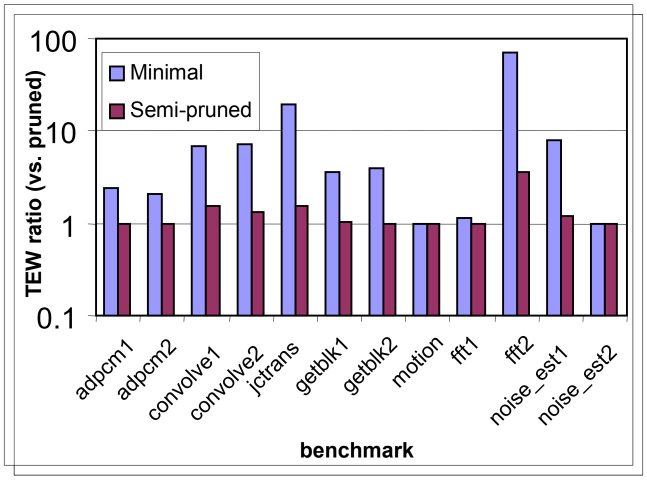


Figure 4: Comparison of total edge weight (TEW) between the minimal and semi-pruned TEW and the pruned TEW

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 4, 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 conclude 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 three algorithms we compared attempt to minimize the number of Φ-nodes, and not the data communication. There is an obvious relationship between the number of Φ-nodes and the amount of data communication. Every Φ-node defines additional data communication, however there can be inter-node data transfer in the absence of 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.

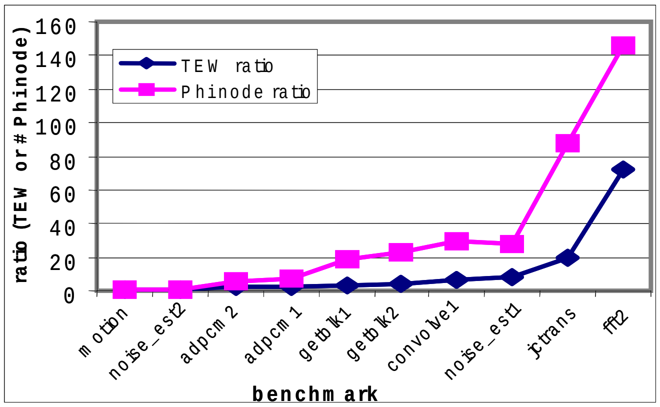


Figure 5: A comparison of total edge weight (TEW) and the number of Φ-nodes using the minimal and pruned algorithms.

In Figure 5, 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 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. We conclude that minimizing the number of Φ-nodes is a good objective function to initially minimize data communication.

Figure 6 charts the total area ratios of the benchmarks. The figure demonstrates that our assumptions about minimizing the TEW have a good correlation with minimizing the area of the circuit. Comparing Figure 4 with Figure 6, you can see that the amount of reduction in TEW correlates with the amount of reduction in total area. For example, the TEW for the benchmark fft2 using the semi-pruned algorithm is approximately 5 times that of the pruned algorithm. A similar result is seen in the total area ratio; the area of the semi-pruned algorithm is about 1.8 times that of the area of the pruned algorithm. Furthermore, the area of the pruned algorithm is almost always the best algorithm in terms of total area. Fft1 is the lone exception, most likely due to its small size and limited number of phi-nodes. Even with this lone outlier, the overall average area improvements are 14% and

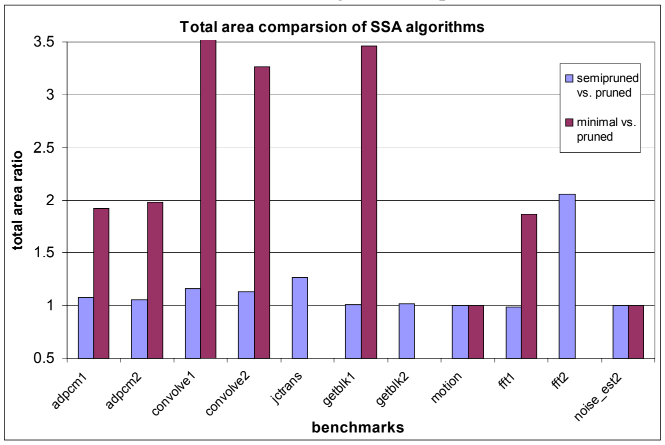


Figure 6: A total area comparison of the benchmarks after logic synthesis. The ratio is the minimal (semipruned) total area divided by the pruned total area. (Omitted benchmarks too large to synthesize.)

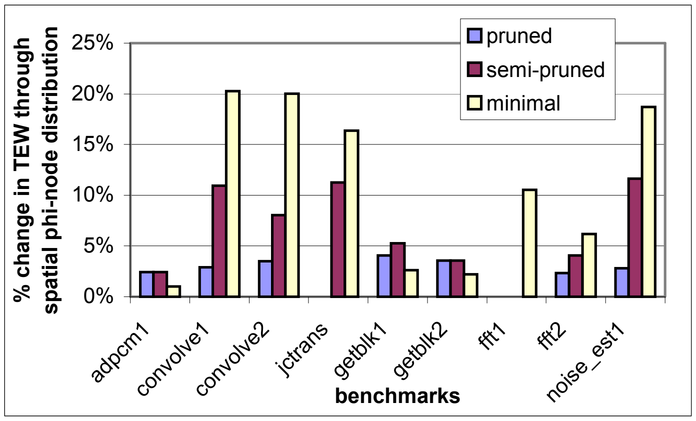


Figure 7: The percentage change in total edge weight when we distribute the Φ-nodes using the three SSA algorithms. (Omitted benchmarks had 0% change.)

87% better using the pruned algorithm over the semi-pruned and minimal algorithm. Thus, the type of SSA algorithm has a huge effect on the area of the circuit implementation. Furthermore, the results indicate that it is worth the small increase in runtime to use the pruned algorithm.

Our next set of experiments focus on using spatial SSA Φ-node distribution to further minimize the amount of data communication. Figure 7 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.

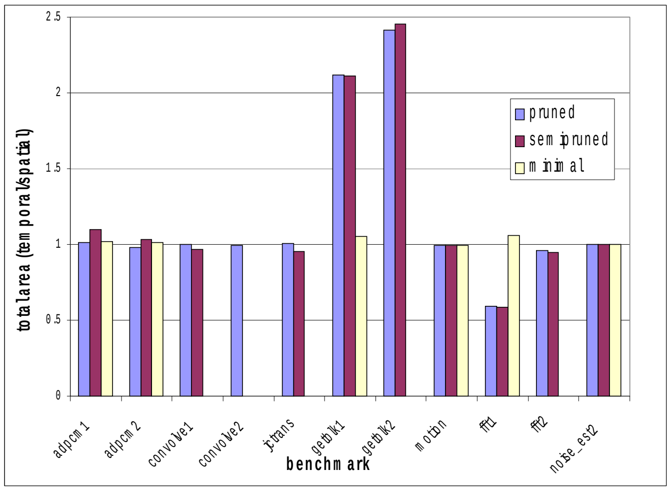


Figure 8: Comparison of the total area of the temporal versus spatial phi node placement for the three SSA algorithms. (Omitted benchmarks too large to synthesize.)

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 8. 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. 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.)

5. Related Work

The past 15 years have brought about a number of platforms that take high-level code and generate a hardware configuration for that platform. We mention a few that are similar in spirit to our framework. 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 framework, as it synthesizes Verilog from C or Fortran.

Several recent reconfigurable system compilers (e.g. [16,17]) use the notion of SSA, though they do not provide any analysis of the effect of SSA on the final circuit. SA-C [18] proposes a single assignment language by definition. It may be possible to use our SSA techniques as a front end to this language.

6. Conclusion

In this paper, we examined the use of SSA within that framework to minimize the amount of data communication between control nodes. We demonstrated a shortcoming of existing SSA techniques when applied to minimizing data communication, as the temporal positioning of the Φ-node is not always optimal. We formulated an algorithm to spatially distribute the Φ-node to minimize the amount of data communication. We showed that this spatial distribution is capable of decreasing data communication (measured as TEW) by 20% for some DSP functions.

In practice, we found that our algorithm frequently increases total circuit area, which is a negative result. Currently we are working on 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 more accurate wire length estimates (obtained during placement. Frequently there is an intermediate range of possible Φ- node placements between the temporal and spatial placements. We intend to explore the possibilities for Φ- distribution across this range. Additionally, we plan to account for the size of duplicated multiplexers. Placement of Φ-nodes will become an algorithmically harder problem, but will yield higher performance through reduced amount of interconnect area.

7. References

[1] R. Cytron et al., "An Efficient Method of Computing Static Single Assignment", *Proceedings of Symposium on Principles of Programming Languages*, January 1989.

[2] P. Briggs et al., "Practical Improvements to the Construction and Destruction of Static Single Assignment Form", *Software Practice and Experience*, 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 et al., "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, Computer Science Department, UCLA, September 2002.

[7] R. Cytron et al., "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 International Symposium on Microarchitecture*, 1997.

[11] M. W. Hall et al., "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,

[13] P. Briggs, T. Harvey and L. Simpson, *Static Single Assignment Construction,* Implementation documentation, 1996.

[14] A. Smith et al., "PRISM II Compiler and Architecture", *Proceedings of IEEE Workshop on FPGA-based Custom Computing Machines,* April, 1993.

[15] J. Babb et al., "Parallelizing Applications into Silicon", *Proceedings of Field-Programmable Custom Computing Machines*, 1999.

[16] J. L. Tripp, P. A. Jackson, and B. L. Hutchings, "Sea cucumber: a synthesizing compiler for FPGAs," Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002

[17] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware," Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002

[18] R. Rinker et al., "An automated process for compiling dataflow graphs into reconfigurable hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 130-9, February 2001.