

ECE497MF Final Report: The Formation and Simulation of a “Whole Program” Gated Singular Assignment Program Dependence Graph

Jeffrey Cook, Derek Gottlieb, Brian Greskamp, Richard Kujoth
Unconventional Computer Architecture Group, UIUC
{jjcook, dgottlie, greskamp, kujoth}@crhc.uiuc.edu

Abstract

The Gated Single Assignment Program Dependence Graph (GSA PDG) provides an executable dataflow representation for imperative programs that might be useful for hardware synthesis. Existing publications on GSA have not attempted to define interpretation semantics for whole programs including side effects (ie. memory accesses and procedure calls). Published works also contain errors even for side-effect-free programs. We have addressed these shortcomings and developed a more complete specification of a GSA form that is able to represent entire programs including side effects. We have also implemented a compiler pass to generate GSA for side-effect-procedures. Finally, we have used simulation to verify our semantics and characterize the parallelism in the GSA graph.

1 Motivation

For reconfigurable computing to gain acceptance, compilers must be able to analyze programs written in imperative languages to effectively identify available parallelism and take advantage of this information when partitioning an application and mapping it onto reconfigurable logic. To this end, the intermediate representation (IR) used by the compiler has a significant impact on its ability to achieve these goals. The gated single-assignment program dependence graph (GSA PDG) [1, 3] appears to have great potential to fill this niche.

The GSA PDG explicitly encodes control- and data-dependencies as directed edges connecting nodes in a single graph. By providing a succinct summary of these program properties, a variety of new compiler optimizations are possible, and many existing transformations may potentially be implemented more efficiently since they can operate on a single graph instead of both control- and data-dependence graphs. Furthermore, the GSA clearly identifies regions for

predicated, speculative execution. Many of these aspects have previously been illustrated for a similar IR [2].

However, previous publications on the GSA have failed to fully specify the graph and the nuances of its construction. As previously specified, the GSA does not support side-effects, subroutines, or unstructured code. This paper seeks to extend the GSA PDG to represent entire programs including side-effects and quantify the parallelism it is able to identify. An overview of the GSA PDG is presented in Section 2 and extensions related to supporting side effects in Section 3. Our compiler infrastructure used to construct the GSA is detailed in Section 4. A discussion of our event-driven GSA simulator and static versus dynamic dataflow interpretations follows in Section 5. Section 6 outlines our preliminary results. Finally, Section 7 presents our conclusions drawn from this work.

2 GSA Overview

Gated Single Assignment form [1] is a dataflow program representation similar to Static Single Assignment form (SSA) except that the graph is fully interpretable using pure dataflow semantics. In SSA, the ϕ node is a *nondeterministic* merge point. Only one of the node's reaching definitions will pass through to the output, but it is necessary to consider the program control flow to determine which. In contrast, control flow is removed from the GSA graph by converting control dependences into data dependences. Each GSA dataflow merge node has additional control inputs to tell the node precisely which reaching definition to select. These control inputs are fed by boolean expressions of the original program's branch conditions. GSA form thus represents the program in pure dataflow terms, with computation nodes connected by token queues. The following section describes the node semantics for eager execution of side-effect-free programs. Later, the semantics are extended to accommodate side effects.

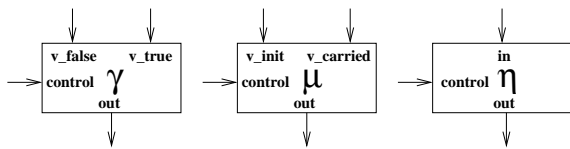


Figure 1. The three special GSA dataflow nodes.

2.1 GSA Nodes

GSA form introduces three new node types, shown in figure 1. The μ and γ nodes replace the ϕ from SSA form. γ nodes govern forward dataflow merges. Wherever two forward control-flow arcs converge on a ϕ node in SSA, GSA places a γ . A single boolean control input tells the γ which of its input tokens to pass along to the output. The γ is strict in its control input. When the control token arrives, it consumes one token from each of its input queues and places the selected token on its output queue. Since the γ has only two data inputs, forward data confluence points with more than two reaching definition are built with trees of γ nodes.

Backward dataflow merges are handled by the μ node. Wherever the SSA form has a ϕ node in a loop header, that node is represented by a μ in GSA form. The μ node serves to gate the loop’s live-in and loop-carried inputs. It accomplishes this with one control input, connected to the output of the loop conditional expression, and one state bit. Initially, the state bit is in state `init`. In this state, it consumes a token from its v_{init} or ‘initial value’ input and places it on the output queue, entering state `iter`. While in state `iter`, the μ is strict in its control predicate. For each `true` token it receives on the control input, it consumes a token from the $v_{carried}$ or ‘loop carried’ data input and places it on the output queue. A `false` token on the control input causes the node to enter the `init` state, awaiting the start of another loop iteration.

Finally, the η node gates the flow of values out of loops. It has a single data input and one control input. The control input is connected to the same loop test node as the loop μ nodes. For each iteration that the η receives a `true` token on its control input, it simply consumes the input data. When it receives a `false` token, signifying that the loop has executed its last iteration, the η consumes a token from the data input and places it on the output queue.

2.2 Node Placement

The original GSA papers state that a loop header must contain a μ node for each variable written inside of the loop that also has a reaching definition from outside the

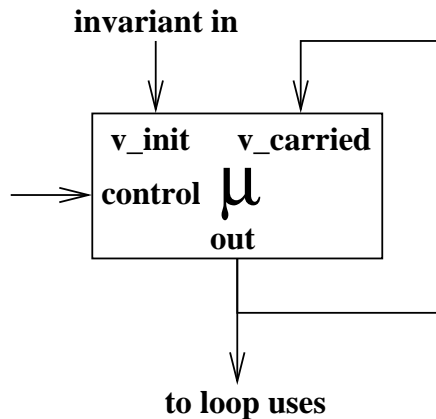


Figure 2. A μ node configured to gate loop invariants into the loop body.

loop. Similarly, an η is needed on each loop exit for every live-out variable that has a μ in the loop header. Trees of γ nodes are placed at forward dataflow confluence points. We will show that these rules are incorrect and give the changes necessary to obtain interpretable GSA graphs for side-effect-free code.

μ Nodes and Loop Invariants The original assertion that μ nodes are only needed for variables written inside the loop is incorrect. In reality, any variable that is read within the loop requires a μ . Since the arcs in the dataflow graph are modeled as token queues, the each execution of the loop body will consume one token from each of the loop body’s input queues. If live-in loop invariants do not pass through a μ before entering the loop, then the first iteration of the loop will read the correct value for the invariant and consume it. Following iterations will either read a wrong token or stall waiting for a token that never comes. The solution is to place a μ node configured as shown in Figure 2 for each loop invariant. Then, the invariant token is consumed only on the first iteration and replicated for all subsequent iterations.

η Nodes for Control Tokens Another fact overlooked in the original papers is that by converting control dependences to data dependences (to drive the control inputs of the GSA nodes), *new* data dependences have been introduced which were not present in the original program. Specifically, it is possible for a γ node following a loop to depend on a token produced inside of the loop. In this case, the γ node’s data input will be connected to either an η gating a value out of the loop or to a loop invariant. In either case, the γ must consume only one input token.

Therefore, the control input must only be allowed to fire when the loop has exited. In other words, the control input needs to be gated by an η . A simpler way to understand this is that the control token produced inside the loop is a data value produced in the loop that escapes to a use outside the loop and hence needs an η .

3 Extending the GSA Form for Whole Programs

The modified GSA form can be extended into a “whole program” GSA form by adding support for (i) memory operations, and (ii) function invocations. In this section, a new semantic is devised to encapsulate these side-effect operations; the new semantic should not add new node types nor should the number of required additional nodes grow anything but linearly in the number of side-effect operations.

3.1 Controlling Side Effects

As previously discussed, the GSA form allows operations to execute before their control dependences are resolved. For side-effect free program graphs, this allows the latency of the control dependence’s computation to be hidden by the early execution of part or all of the data-path that was control dependent on it; a potentially negative aspect of this execution, however, is that the operations down the not-taken path are also speculatively executed, requiring more computations to be performed. These additional computations do not affect the program’s correctness, however, and thus can be acceptable. If one or more of those speculatively executed operations has a side-effect, then all bets are off and program correctness can be negatively affected.

In our study to create a whole program GSA form, three cases with side-effects were identified. The first is related to the original GSA form; if the not-taken path leading into a γ -node contains an infinite loop, as depicted in Figure 3, the loop will be evaluated ad infinitum; depending on the interpretation semantics, this could cause the program to never terminate even though all live-out values were correctly computed. The second kind of side-effect (the obvious case) is loads and stores to memory; two constraints must be placed on such operations: (i) all loads and stores are partially ordered on a may-alias / do-alias basis, as illustrated in Figure 4, and (ii) a store can not execute/commit until its control dependences have been resolved and it is on the taken path. The third kind of side-effect relates to function calls; since the two previous kinds of side-effects (infinite loops and memory operations) can be present in a function, the same care and precautions must be taken for function calls as the previous two cases.

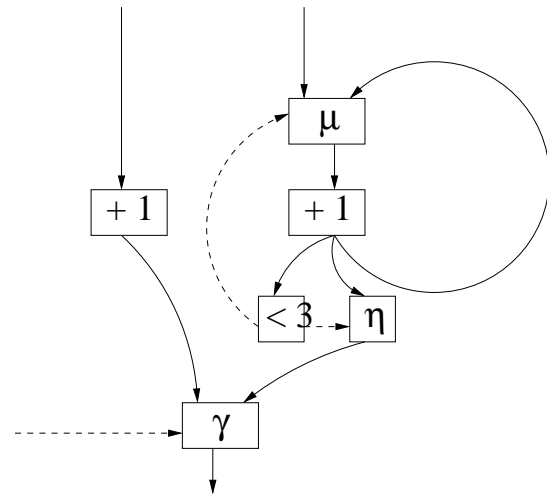


Figure 3. Potential Infinite Loop

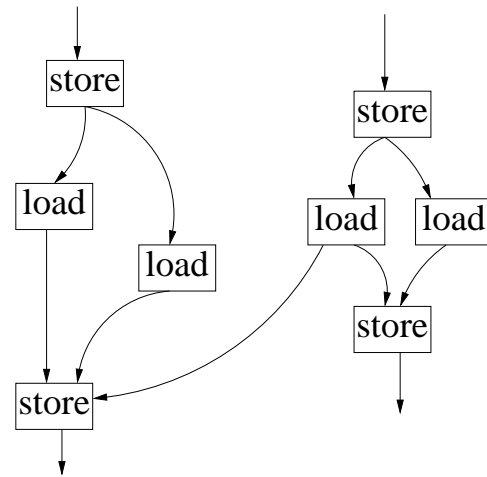


Figure 4. May/Do-Alias Partial Ordering

The above three cases for side-effects can be handled succinctly and elegantly through a single semantic; as was alluded to previously, each side-effect prone operation should not execute until all of its control dependences have been resolved and it has been determined that the operation is on the taken path. The naïve approach is to form an η -node gating tree for each side-effect operation, where each η -node in the tree is controlled one of the operation’s control dependences; this approach, of course, would cause an exponential growth of nodes in the nesting depth of the code, a very unattractive result. Instead, an only linear overhead approach is to create a single pair of η -nodes (one η_T and one η_F) per control dependence, as shown in Figure 5. This results in a single gating tree for the whole program/function; the input to each η -node is defined as the

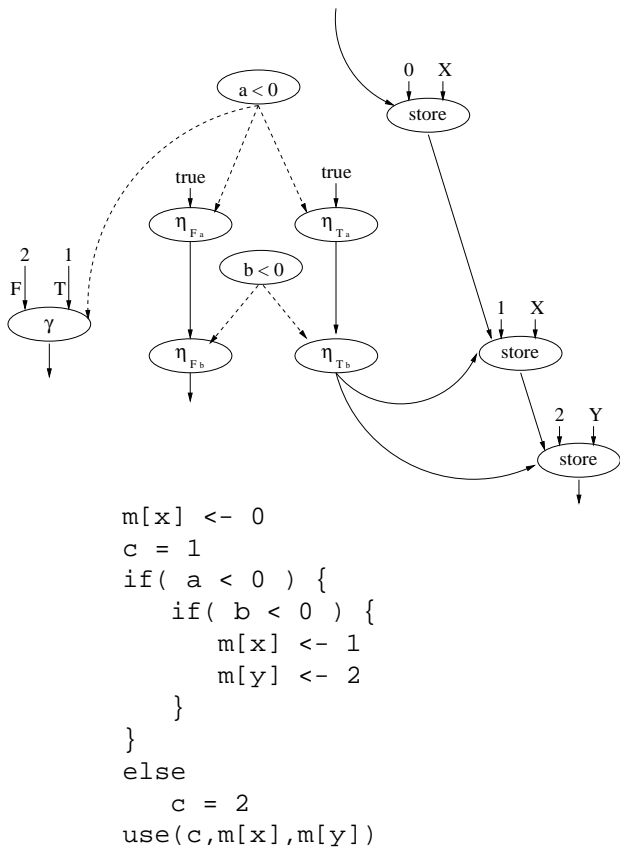


Figure 5. Eta Tree for Above Example Code

output of the previous control dependence’s η -node, and a `true` token feeding the root of the tree. Whenever a copy of this `true` token arrives at a side-effect node through the gating tree, the operation may be executed when its remaining inputs are available. For the remainder of the paper, this gating tree will be referred to as the *side-effect gating tree* and the immediate control dependence gating η -node as the *side-effect gate*.

3.2 Refining and Defining Node Semantics

Given the newly constructed gating tree for side-effects, the semantics for loops, memory operations, and function calls can be refined and defined as follows:

3.2.1 Loops

As a loop may never terminate, it must be gated by the side-effect gate; fortunately, the initial iteration of the loop can be allowed to execute so as to potentially hide the latency of the local control dependence computation controlling the side-effect gate. The iteration of the loop, however, must stall until the side-effect gate evaluates. Thus, the μ -

node’s semantics have been modified to not transition into the `iter` state unless the side-effect gate value returns true to the *on_path* port; if true is returned to the *off_path* port, the iteration value sitting on the $v_{carried}$ port as well as the *control* port token is consumed. This is pictured in Figure 6.

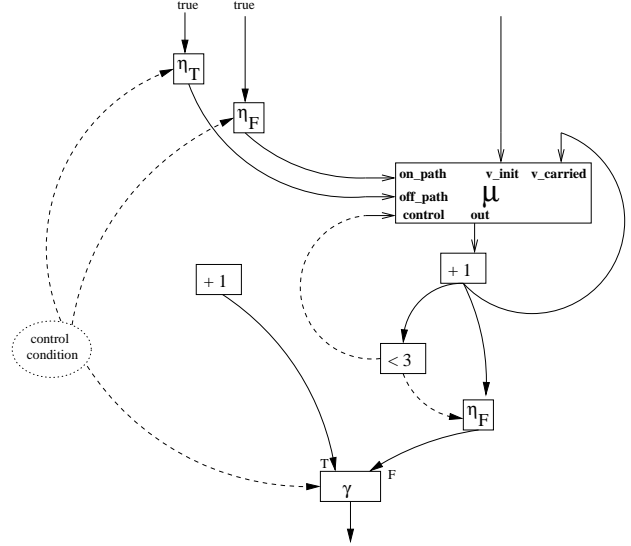


Figure 6. Revised μ Semantics

3.2.2 Memory Operations

Both load and store nodes (Figure 7) have two ports that enforce a partial ordering; the dep_{in} port requires a single `true` token that signifies that the node’s immediate dependences have executed and it may fire when its remaining inputs are valid; the dep_{out} port distributes `true` tokens to all of the nodes immediate dependents once the node has fired. The $address$, $data_{in}$, and $data_{out}$ ports work as expected. As with all side-effect operations, the local side-effect gate must gate the node by activating either the *on_path* or *off_path* port, (at least for store nodes) so that memory is not corrupted by spurious stores.

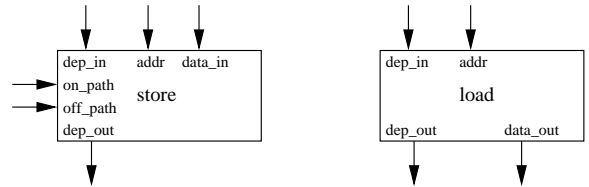


Figure 7. Load and Store Nodes

In Figure 8, one can see the η -nodes gating the dep_{in}

port using the tokens from the *dep_out* ports of the load operations, so as to allow multiple depends-on dependencies.

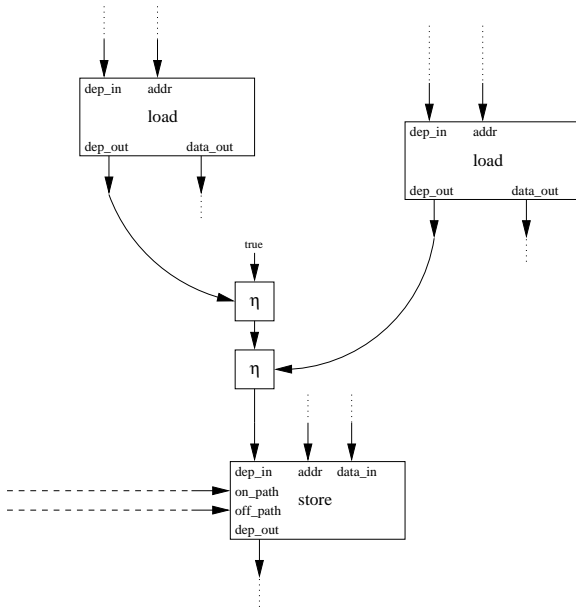


Figure 8. Memory Dependencies Gating

3.2.3 Function Calls and Returns

At first glance, adding function calls and returns to the GSA seems rather straightforward by adding three new nodes (Figure 9): (i) a *call* node that summarizes all function input parameters, which is also gated by the local side-effect gate, (ii) a function *start* node that provides one or more inputs to the graph it heads, and (iii) a *return* node that returns a single token to the calling context. The return node, however, is exactly where it gets interesting; up to this point, the modified GSA graph could still be static data-flow interpreted (as discussed later), but now the return node needs to return the token to one of many possible callers.

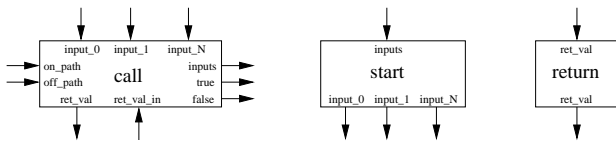


Figure 9. Function Call and Return Nodes

There are several “solutions” to this problem; one solution is to assume the interpreter will somehow “tag” the outputs from the start node so that the input token to the return node will also have that tag; when firing the return node, the interpreter would then “know” where

to send the token to. This approach has the advantage that there can be only one return node per function, and that the “compilation” of the function’s graph can take place independently of the callers; this approach, however, breaks the standard static data-flow interpretation by having additional information carried by tokens. A perhaps better solution is to take advantage of “link-time” processing; at link-time, a sea of η -nodes would be generated, one for each caller node. The η -node for a particular caller then gates with a *true* token the path from the output of the return node to the return value input of that calling node; to ensure the remaining η -nodes consume their values without output, the caller node must also broadcast a *false* token to the remaining η -nodes. This is pictured in Figure 10. Other solutions and variations of these solutions exist, but are not further discussed.

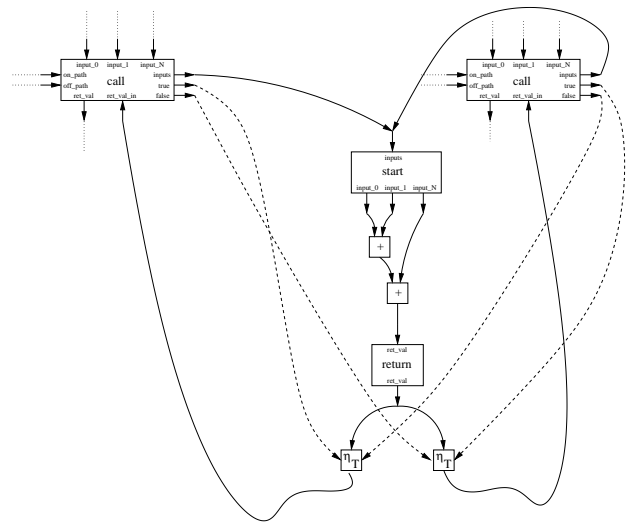


Figure 10. Function Call and Return Example

If one was keen, one might observe that the inputs to start and these η -nodes are not single assignment, unlike all the constructions presented thus far that painstakingly build gating trees to enforce single assignment; for a function header start node, the notion of non-single assignment inputs is, of course, completely natural and can not be avoided without moving the non-single assignment off to other nodes. The return’s η -nodes, on the other hand, now have the non-single assignment property; given that this case arises only during a function return, the η -nodes can still be assumed single assignment for any compiler analysis passes by special casing or just by not analyzing the η ’s of return nodes, or by simply creating a new node type.

4 GSA Implementation

The first attempt at an algorithm for GSA construction appeared in [1], along with the original description of the GSA form. Unfortunately, that paper had numerous flaws, largely because the algorithm they set forward was never implemented or tested. Subsequently, Paul Havlak [4] set out an algorithm for constructing the GSA graph incrementally from SSA form. Our algorithm closely parallels his. We have implemented the algorithm in the Machine SUIF compiler infrastructure and compiled a small number of test cases to verify correct handling of nested loops and exceptional exits. Although some debugging remains before we can certify that the pass will process all side-effect free programs, we have high confidence in the algorithm from the tests that we have executed.

From experience, we expected to discover problems and inconsistencies in the published GSA construction algorithms during implementation, so we opted for an inelegant but easy-to-modify stepwise approach to GSA construction. The first phase preconditions the control flow graph. The second phase is SSA construction, for which a Machine-SUIF implementation is already available. The final phase converts the SSA graph to GSA through a series of many small steps. This approach proved helpful during implementation when the problems of section 2.2 were discovered. Each was fixed by simply adding another step in the SSA \rightarrow GSA conversion process.

4.1 Implementation Details

Like many compiler transformations, the GSA conversion pass requires that the program control flow graph be reducible. No generality is lost since any program can be converted to a reducible form through node splitting, although code expansion may be exponential in pathological cases. Since an important goal of this work is to eventually convert whole programs to GSA form, we take a brief digression to discuss the node splitting problem and a simple solution implemented in Machine SUIF.

4.2 Node Splitting

Irreducible code poses a problem for many compiler optimizations, including data flow analysis and loop transformations. Code being irreducible means that it includes a subgraph in which there is a cycle between nodes, none of which dominate each other. An example of an irreducible flowgraph can be seen in Figure 11. Hecht and Ullman [5] devised a method of finding these irreducible sections of code and correcting them through duplication.

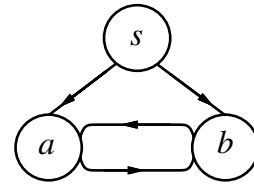


Figure 11. A basic irreducible loop. [6]

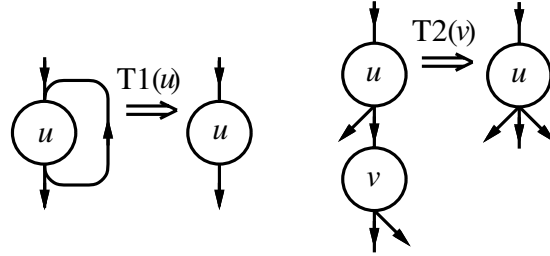


Figure 12. The T1 and T2 transformations, used to determine the reducibility of a graph. [6]

Hecht and Ullman's method of detecting irreducible regions involves two types of transformations to the code. Transformation 1 (T1) rids nodes of self-loops. Transformation 2 (T2) combines child nodes with their parents when they only have a single parent, or predecessor. Examples of these transformations are illustrated in Figure 12.

These operations are performed repeatedly until they cannot be performed any more on the graph. If the resulting graph is a single node, then the graph is reducible. If not, there is an irreducible region within the graph. A third transformation (T3) is provided to make the graph reducible. T3 finds nodes that have multiple predecessors and duplicates them, resulting in a copy of the original node for each predecessor (illustrated in Figure 13). This process of alternating between performing T1/T2 operations and the T3 operation is repeated until a single node is left. The graph is now reducible. Using heuristics, this method can be optimized and some of the splits can be avoided or reordered to generate a smaller output code size, but this is a basic method of making a graph reducible.

An implementation of this algorithm needs to be able to reverse all of the T1 and T2 operations that have been performed on the original graph, but leave the T3 operations products in the graph to result in a reducible, executable version of the graph. Our implementation, in Machsuiif, has the same effect, but does not need to reverse any operations that it performed to make the graph reducible.

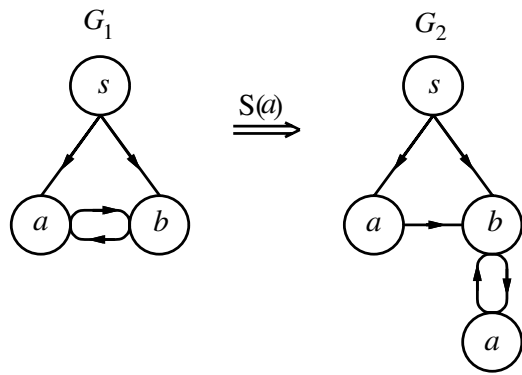


Figure 13. The T3 transformation, used to transform irreducible loops into reducible loops by node-splitting. [6]

Our implementation performs the T1 and T2 operations on a copy of the original graph and performs the T3 operations on both graphs. So, when the algorithm has reached its end, with a single node left, the modified original graph remains and is reducible.

4.3 Preconditioning the CFG

After node splitting, the GSA pass ensures that each loop has a unique header (*i.e.* that nested loops do not share headers). In the case that shared headers are found, the pass adds empty nodes to the CFG to serve as headers for the inner loops. Additionally, it checks for loop headers with multiple forward in edges. If it finds any such headers, it creates an empty CFG node preceding the header and connects all of the header's forward in-edges to the preheader. Together, these transformations ensure that after SSA conversion, each loop will have a set of ϕ nodes in its header with exactly one forward input and one backward in-edge. The loop preheader will contain ϕ nodes with forward in-edges only. Later, it will be trivial to map these ϕ nodes to μ and γ nodes, respectively.

Although we have transformed the loop headers, we have so far left the bodies and exits untouched; loops are allowed to contain arbitrary `break` and forward `goto` statements, as well as `continue` statements. The only transformation that affects the loop tail is the conversion of all pre-test loops to post-test loops with an 'if-do-while' structure. Finally, all branches in the program are converted to BTRUE statements by modifying the conditional expressions appropriately. This branch conversion simplifies gating path construction.

4.4 Generating SSA Form

Since each γ tree and μ node in GSA form corresponds to an identically-placed ϕ node in SSA form, and since most compilers already have an SSA-conversion pass, we will make use of SSA as a starting point for GSA generation.

After the CFG has been preconditioned, any algorithm can be used for SSA conversion. The resultant SSA form will contain ϕ nodes for all γ trees and some μ nodes, but not for any η nodes because η nodes are not associated with dataflow confluence points. Therefore, η nodes are added explicitly by splitting loop exit edges and inserting ϕ nodes for each value that will need an η . For each exit, those variables that need η nodes are the variables assigned within the loop (*i.e.* those for which the loop header has a ϕ node). Next, all subsequent uses are updated to point to the newly created ϕ nodes.

Finally, as discussed in Section 2, μ nodes are needed not only for variables that are loop-carried, but also for those that are loop-invariant. SSA form will not have ϕ nodes in the loop header for invariants, so they must be inserted. Any variable that is read within the loop (including loops nested inside of the current loop) needs a ϕ in the loop header, so one is inserted if it does not already exist. All uses within the loop are updated to point to the newly created ϕ .

4.5 Converting SSA to GSA

Now the SSA graph contains a ϕ node for almost¹ every GSA node in the final graph. All ϕ nodes in CFG blocks with two or more forward in-edges will become γ trees. Those in blocks with a backward in-edge become μ nodes, and with one in-edge become η nodes. What remains is to construct the boolean expressions that will control each of these nodes' control inputs. Fundamental to this computation is the notion of a *gating path* G_n , a boolean expression of branch decisions which is true if and only if control reaches n from $idom(n)$.

Although it is possible to compute all of the gating paths in the program in $O(nlg(n))$ time using path compression [8], we use a much less efficient algorithm that computes the gating path for a given phi on demand. Given a node n and its immediate dominator d , we simply enumerate all paths from $n \rightarrow d$. Each path comprises a string of branch decisions that are used to determine when a node should fire and which inputs it should select. Gating path construction is invoked for each ϕ as it is converted. Conversion of each node type proceeds in order as follows.

μ and η Conversion The μ node takes the loop exit condition as a control input. For many loops, this is simply

¹Some γ nodes may still require η nodes on their control inputs.

the loop conditional, but for those with exceptional exits, the exit condition is more complex. In either case, a boolean expression for the exit condition can be found by computing the gating path for the loop tail. The loop exit expression is the gating expression for the loop tail logical-anded with the condition tested in the tail. A gate tree of AND, OR, and NOT operations implementing the loop exit expression is constructed and connected to the μ node's control input. All η nodes simply have their control inputs connected to the same gate tree as the μ nodes in their loop's header.

γ Conversion The ϕ nodes in SSA form tell which assignments reach the ϕ for each of the node's control-flow in-edges. For each forward dataflow confluence point, we must find the gating path from the immediate dominator of the ϕ to each of its immediate predecessors. For each immediate predecessor, the union of all gating paths to that predecessor gives a boolean expression that is true when that input to the ϕ should be selected. Having obtained a gating expression for each predecessor, we can build a tree of two-input multiplexers to select the appropriate reaching definition, where the mux control inputs are a subset of the branch decisions that appear in the gating expressions. These two-input multiplexers are in fact γ nodes.

η Nodes for Control Tokens Just as tokens that escape from loops need η nodes to ensure that they only become available to the rest of the graph when the loop exits, control tokens (*i.e.* those tokens that reach the control inputs of GSA nodes) may also escape from loops and therefore require η nodes. These extra nodes are inserted in the last phase of construction for any γ node where the instruction that produces the control token is in a different loop or a different nesting level than the instructions that produce the two data tokens.

4.6 A Simple Example

This section covers a simple example of GSA formation starting with the program CFG in Figure 14. After pre-header insertion (not shown), the program is converted to SSA in Figure 15. Here, dataflow arcs replace control-flow arcs. Next, μ and η nodes are converted in Figure 16. A tree of gates has been generated to calculate the loop exit expression. The output of t_7 will be true if the loop should continue for another iteration or `false` otherwise. Note how the loop exit expression $(t_2 + \bar{t}_2 \bar{t}_3) t_4$ can be obtained by taking the union of the gating expressions for the loop tail's two in-edges ANDed with the result of the branch decision t_4 . The loop exit expression controls both the loop μ and the η . Data arcs that are used only to control the GSA nodes are illustrated with dotted lines.

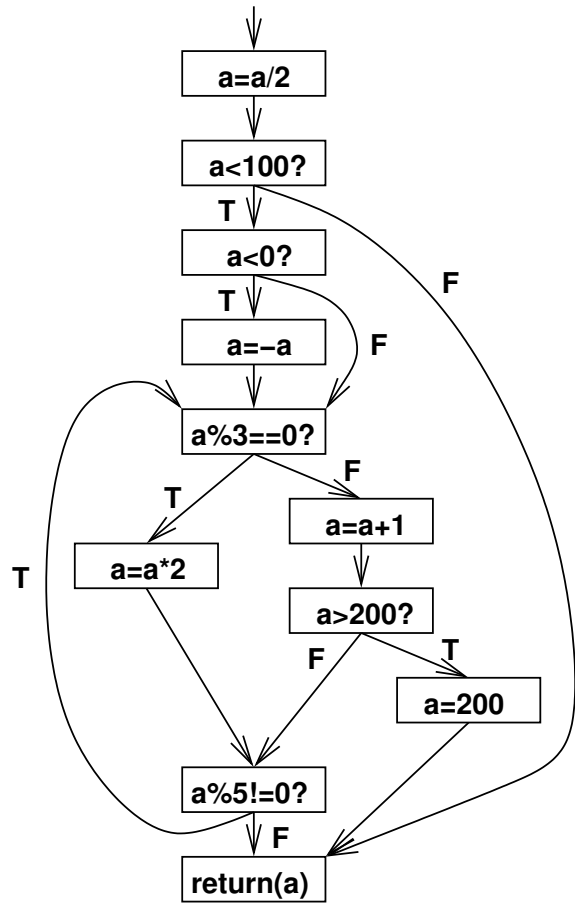


Figure 14. Original program CFG.

Figure 17 shows the conversion of γ nodes. A γ tree is constructed for the ϕ at `a9`. The γ at `a11` passes either the loop-carried value of `a` or the value of `a` assigned in the exceptional loop exit if the exceptional exit was taken. The graph as it appears after γ -conversion is actually incorrect, however. The control token for the γ at `a12` is produced inside of the loop, while both of the data inputs are produced outside the loop. Therefore, the control input to the γ must be gated with an η predicated on the loop termination condition as in Figure 18.

5 Interpreting the GSA

In order to verify the correctness of our GSA implementation, we developed an event-driven simulator using an abstract token-based model of computation and supports both static and dynamic dataflow semantics. Simulation begins by parsing a textual graph description consisting of a list of GSA nodes and edges, and adding all of the constant nodes onto the event list for the next cycle. Each cycle, the

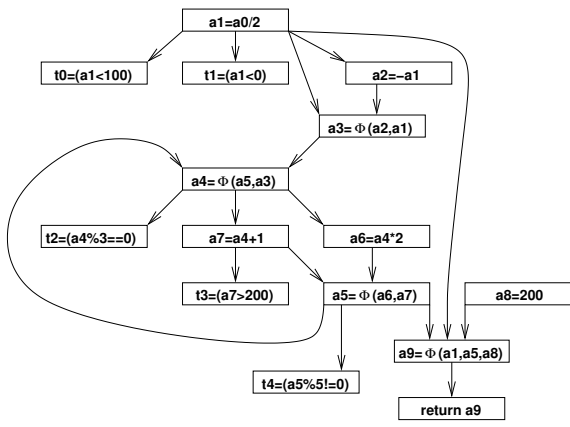


Figure 15. After conversion to SSA. All arcs are data flow, not control flow.

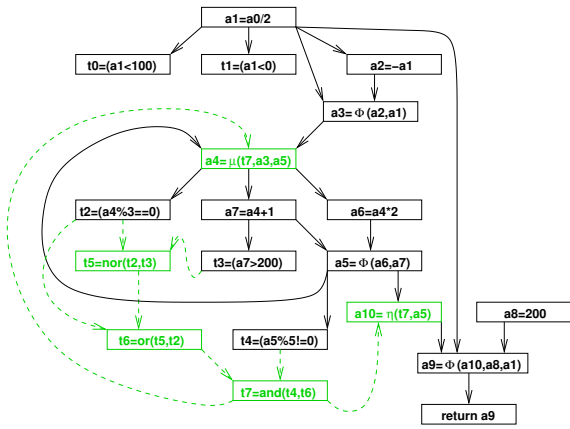


Figure 16. After converting μ and η nodes.

nodes added to the event list during the previous cycle are evaluated. If valid tokens exist on all input ports of a node during evaluation, the node consumes a token from each input port (modeled as queues), performs the corresponding computation, and places corresponding tokens on the input ports of its fan-out, adding fan-out nodes to the event list for the next cycle. When all events for a given cycle have been handled, the simulator transitions to the next cycle and proceeds to evaluate the next event list. This process continues until a cycle is reached where no new events have occurred.

The sole exception to this evaluation model is the μ node, which highlights the key distinction between static and dynamic dataflow semantics. Under static dataflow semantics, parallelism is limited by the requirement that only a single instance of any node may exist at a time. The GSA operates under this restriction by ensuring that only a

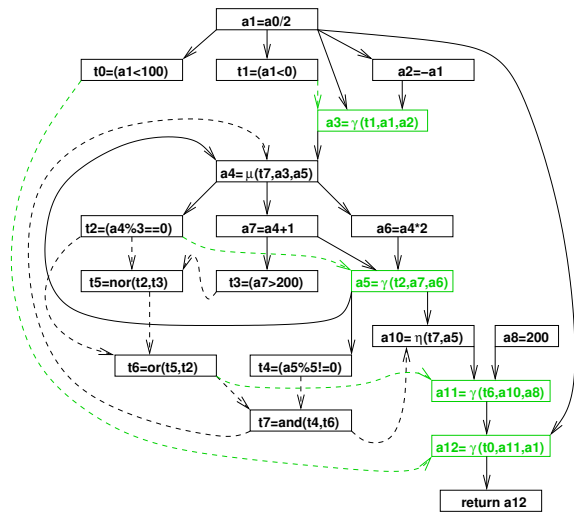


Figure 17. After converting γ nodes.

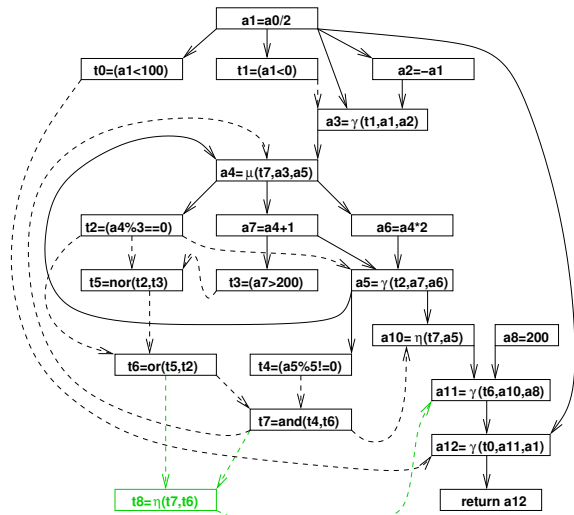


Figure 18. Completed GSA form after inserting η for γ control path.

single instance of a loop may exist at a time. The first token to arrive at the v_{init} port is passed to the output and causes a state transition within the μ node, preventing additional tokens from being consumed from the v_{init} port. For each subsequent true token that arrives at the μ 's control port, a corresponding data token from the $v_{carried}$ is consumed and passed to the output. A false token (signifying the termination of the loop) results in another state transition, which allows the μ node to consume tokens from the v_{init} port and initiate a new loop instance.

This semantic may be extended to support dynamic dataflow semantics, wherein multiple loop instances may execute in parallel. One such extension replaces the state bit in the μ node with a counter that represents the loop instance. Each time a token is consumed from the v_{init} port, this counter is incremented and the resulting output token is tagged with the counter value. Nodes then consume tokens that all have the same loop tag associated with them, which is possible since all loop inputs and loop invariant values are gated by μ nodes. In order to support nested loops, these counter values must actually be maintained as a stack. When values pass through a μ node, a counter value is pushed onto the token's tag stack. When a value exits a loop via an η node, this stack is popped to return to the previous loop tag.

5.1 Example: N-primes

Figure 20 illustrates the GSA graph generated for the C subroutine `nprimes()` (Figure 19), which calculates the number of prime numbers that are smaller than or equal to an input `n` (represented by the **REG** node in the Figure 20).

This graph was largely constructed using our compiler infrastructure and verified using our event-driven simulator under static dataflow semantics. Statistics gathered include run-time estimates, number of instructions/nodes executed, and measured ILP. These are presented in Section 6.

6 Results

As mentioned in Section 5.1, we have compiled and simulated a GSA graph for the N-primes function that counts the number of primes less than or equal to 50. In this simulation, we have assumed that a single cycle latency for all nodes, and that there is sufficient hardware for any possible execution (*i.e.*, unrestricted queues on the input ports and unlimited resources for multiple simultaneous instances in the dynamic dataflow model). These results are somewhat pessimistic since we are not currently generating optimal GSA graphs and η nodes would not necessarily have to incur a cycle delay if they were implemented as registers.

```
int nprimes(int n)
{
    int retval;
    int i, j;
    int is_prime;

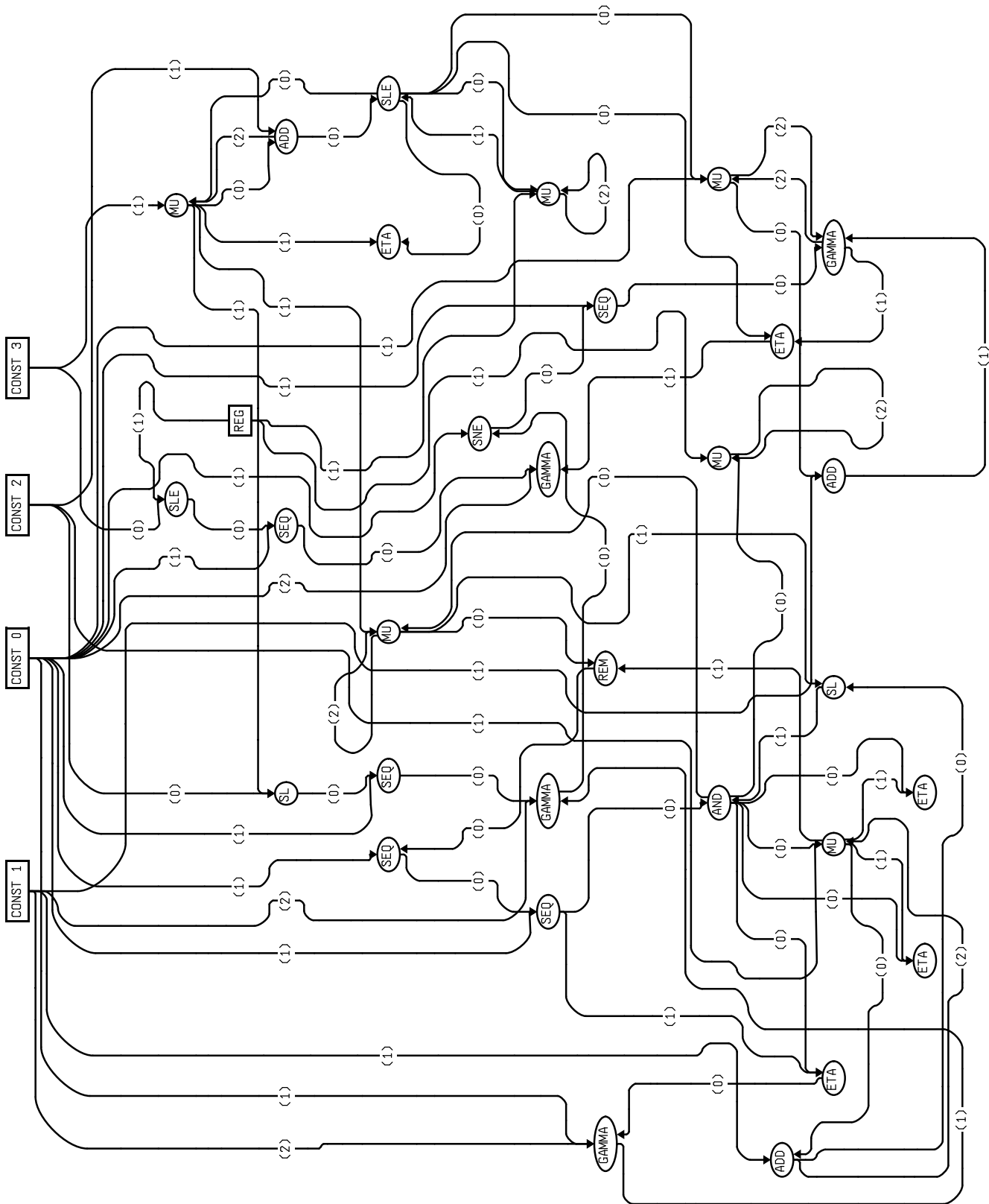
    retval=0;
    for(i=3;i<=n;i+=2) {
        // Is i prime?
        is_prime=1;
        for(j=2;j<i;j++) {
            if(i%j==0) {
                // No, i isn't prime
                is_prime=0;
                break;
            }
        }
        if(is_prime) {
            retval++;
        }
    }
    return(retval);
}
```

Figure 19. N-primes function

Using static dataflow semantics, the number of primes less than or equal to 50 may be computed in 449 cycles with a total of 1075 nodes firing. Furthermore, we achieve a maximum of 6 nodes firing in a single cycle with an average of 2.39. The parallelism may be increased by using dynamic dataflow semantics, which would allow multiple instances of the inner loop to execute concurrently. Exploiting this increases the average nodes per cycle to 8.56 with a maximum of 19 nodes firing in a single cycle. This would result in a similar reduction in execution time at the cost of additional hardware.

As the process for creating executable GSA representations of entire programs has not yet been fully automated and we only have simulated results for the N-primes problem, we developed some rules for a Lam and Wilson-style [7] simulation of our benchmarks in a GSA model and compared the resulting ILP vs. the ILP attained using some of the Lam and Wilson models, namely CD-MF and the ORACLE machine.

We performed our simulations using a modified version of the trace-based simulator that we created for HW2. Each instruction is modeled as a taking a single cycle to execute. The rules for the GSA model were very similar to those of the CD-MF model from Lam and Wilson, except that side-effect free instructions could execute preemptively, when their data dependencies' control dependencies were



Created with aiSee V2.0 (ERP-Version) (c) 2000 AbsInt Angewandte Informatik GmbH. Commercial use prohibited!

Figure 20. N-primes function (GSA)

Benchmark Name	BASE cycles	ORACLE cycles	CD-MF cycles	GSA cycles	ORACLE ILP	CD-MF ILP	GSA ILP
AES	1106827	34399	40589	37410	32.2	27.3	29.6
FIR	1499079	6679	6693	6689	224.4	224.0	224.1
MPEG1	4756233	196616	197104	197031	24.2	24.1	24.1
TSP	3909895	659064	991522	827569	5.9	3.9	4.7

Table 1. Comparison of the parallelism for the ORACLE, CD-MF, and GSA models.

resolved. This simulates the predicated execution of code in our GSA representation of the program. Data dependencies must be obeyed, but different flows of control can be executed preemptively, the correct path being selected later in the execution.

The results of this study are located in Table 1. We performed our simulations on versions of AES encryption, a 50-tap FIR filter, a MPEG1 I-frame encoder, and the Traveling Salesman’s Problem (TSP) on a search space of 8 cities. The GSA model of execution’s execution time consistently lies between those of the ORACLE machine and the CD-MF model.

GSA’s advantage over the CD-MF model of execution reduces execution time when instructions’ data dependencies lie behind one or more control dependencies. Benchmarks with more unique control-flow patterns tend to exploit this advantage more than benchmarks with simple loops. Examples of this are the TSP and AES benchmarks. Overall, the GSA model of execution performs about 6% faster than the CD-MF model, while ORACLE performs about 7% faster than GSA on the four benchmarks that we have included.

These results show that a GSA model of execution has the potential for great performance benefits on certain applications, thus getting execution times down closer to the unattainable ORACLE machine. When GSA doesn’t yield much to any performance benefit over a CD-MF model, GSA is not worthwhile, since a lot of additional hardware is necessary to execute it.

7 Conclusions

We have presented revisions to the original GSA specifications which allow graphs for side-effect-free programs to be interpreted correctly. We have also proposed extensions of the form to allow representation of whole programs including side effects. Section 4 described a compiler pass for generating the side-effect-free graphs, and section 5 discussed the simulation methodology used to verify correctness of one sample graph. While much work remains to be done debugging the compiler and verifying its functionality on larger procedures, we have a high degree of confidence that we have solved at least two problems with

the original side-effect free formulations. A set of semantics for side-effect execution has been proposed in section 3, although we have not tested their correctness either formally or experimentally. Finally, we have examined the effects of GSA’s inherent predication on *ILP* and cycle counts in section 6. Much work remains to be done toward the ultimate goal of using a GSA-like language for hardware synthesis.

References

- [1] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, 1990.
- [2] M. Budiu and S. C. Goldstein. Pegasus: An Efficient Intermediate Representation. Technical Report CMU-CS-02-107, CMU, May 2002.
- [3] P. L. Campbell, K. Krishna, and R. A. Ballance. Refining and Defining the Program Dependence Web. Technical Report 93-6, University of New Mexico, March 1993.
- [4] P. Havlak. Interprocedural Symbolic Analysis. Technical Report TR94-228, 17, 1997.
- [5] M. Hecht and J. Ullman. Flowgraph Reducibility. *SIAM J. Comput.*, 1(2):188–202, 1972.
- [6] J. Janssen and H. Corporaal. Making Graphs Reducible with Controlled Node Splitting. *ACM Trans. on Prog. Lang. and Systems*, 19(6):1031–1052, 1997.
- [7] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proc. of the Intl. Symp. on Comp. Arch.*, May 1992.
- [8] P. Tu and D. Padua. Efficient building and placing of gating functions. *PLDI*, pages 47–55, 1995.