Static Single Assignment Form and the Dominance Relation

Philip Brisk  
Processor Architecture Laboratory  
Swiss Federal Institute of Technology (EPFL)  
Lausanne, Switzerland  
{philip.brisk}@epfl.ch

Majid Sarrafzadeh  
Department of Computer Science  
University of California, Los Angeles (UCLA)  
Los Angeles, CA 90095  
majid@cs.ucla.edu

ABSTRACT
A procedure is defined to be strict if every variable is defined before it is used along every path of program execution. A regular program is a strict procedure in Static Single Assignment (SSA) Form. Recently, it has been proven that the interference graph for regular program is a chordal graph. This yielded an optimal polynomial-time algorithm for register allocation for high-level synthesis. Unfortunately, for SSA Form applications that are not regular, the interference graph is not chordal. To address this issue, this paper presents an algorithm that can take an irregular SSA Form procedure and convert it into regular form. This makes it possible to optimally allocate registers for any procedure in SSA Form, regardless of whether or not it is regular. In other words, optimal polynomial-time register allocation can be achieved for any procedure by converting it to SSA Form.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors – compilers, optimization.

General Terms
Algorithms, Performance, Languages

Keywords
Static Single Assignment (SSA) Form, Strict Program, Regular Program, Dominance, Immediate Dominator, Dominator Tree.

1. INTRODUCTION
In high-level synthesis, the problem of register allocation—i.e. determining the minimum necessary number of registers to allocate to a design—is equivalent to computing the chromatic number of an interference graph. Since graph coloring is NP-Complete in the general case, we would like to see that register allocation is also NP-Complete; however, this is not always the case. The reason is that algorithms can be constructed in such a manner that the interference graph belongs to classes of graph whose chromatic number can be computed in polynomial time. In particular, a procedure represented in Static Single Assignment (SSA) form has a chordal interference graph [1, 3, 11-12], which can be colored optimally in \(O(|V| + |E|)\) time [8].

Let an operation \(v \leftarrow ...\) that writes a value to variable \(v\) be called a definition of \(v\), and any operation \( ... \leftarrow v\) that uses the value written to \(v\) be a use of \(v\). In SSA Form, every variable \(v\) is defined once, and every use of \(v\) corresponds to its definition. Variable \(v\) is defined to be live at point \(p\) in a program if there is path from a definition of \(v\) to \(p\), and a path from \(p\) to a use of \(v\).

In this paper, we assume that pruned SSA Form is used, which inserts \(\varphi\)-functions (see Section 3.3) into the application only at points where variables are live.

A procedure is defined to be strict [4] if every variable is defined before it is used on every possible path of program execution. Languages such as Java impose strictness on every variable as part of the syntax definition. Other languages, such as C/C++, permit non-strict programs. A regular program is defined to be a procedure in SSA Form that is also strict.

There are several definitions of a chordal graph; here, only one is needed: \(G = (V, E)\) is a chordal graph if and only if \(G\) is the intersection graph of some set of subtrees of a tree.

The live range of variable \(v\) is the set of all program points where \(v\) is live. Two variables interfere if the intersection of their live ranges is non-empty. If \(V\) is the set of variables in a procedure, then \(G = (V, E)\) is an interference graph if an edge \(e = (u, v)\) is added between every pair of variables that interfere.

It is easy to see that the live range of each variable corresponds to a subtree of the dominator tree (Section 3.2) of a procedure. The following theorem follows immediately:

**Theorem 1.** Let \(P\) be a regular program. Then the definition point of variable \(v\) dominates each use of \(v\) [4] as well as every point in the program where \(v\) is live [1, 3, 11-12].

There are several definitions of a chordal graph; here, only one is needed: \(G = (V, E)\) is a chordal graph if and only if \(G\) is the intersection graph of some set of subtrees of a tree.

The live range of variable \(v\) is the set of all program points where \(v\) is live. Two variables interfere if the intersection of their live ranges is non-empty. If \(V\) is the set of variables in a procedure, then \(G = (V, E)\) is an interference graph if an edge \(e = (u, v)\) is added between every pair of variables that interfere.

It is easy to see that the live range of each variable corresponds to a subtree of the dominator tree (Section 3.2) of a procedure. The following theorem follows immediately:

**Theorem 2.** Let \(P\) be a regular program. Then the interference graph for \(P\) is a chordal graph [1, 3, 11-12].

Brisk et al. [3] recognized that Theorem 2 effectively states that register allocation—in the context of synthesis—has a polynomial-time solution for regular programs. Since there is an \(O(|V| + |E|)\)-time algorithm for coloring chordal graphs [8]:

**Theorem 3.** Let \(P\) be a regular program with interference graph \(G = (V, E)\). Then register allocation can be solved optimally with time and space complexity \(O(|V| + |E|)\).

Theorems 1-3 and are true only for regular programs. This paper presents an algorithm to convert an SSA Form program that is not regular into a regular program. This, in turn, ensures that register allocation can be solved optimally for any procedure by converting it to SSA Form. Once in SSA Form, various
optimizations and transformations applied to the procedure may cause it to become irregular. As shown in Section 6, many procedures in common C-language benchmarks are irregular when converted to SSA Form. By using the algorithm presented in this paper, it becomes possible to optimally allocate registers for any application in SSA Form, regular or not.

1.1 Brief Digression
Although it is somewhat beyond the scope of this paper, register allocation in the context of compilers remains NP-Complete [1, 11], even for chordal graphs. In a software compiler, the number of registers in the target architecture is fixed. Hence, computing the chromatic number of an interference graph does not tell us which variables to spill. Likewise, the assignment of registers to variables with the goal of minimizing the number of copy instructions inserted into the application is also NP-Complete for chordal interference graphs [19].

2. RELATED WORK
2.1 Register Allocation in Synthesis
Register allocation in the context of high-level synthesis has been studied by many researchers, and many of the techniques are 20+ years old. The first to model register allocation as a coloring problem were Tseng and Stieweck [23], in 1986. Specifically, they modeled the problem as computing a clique partition of a compatibility graph—the complement of an interference graph; a clique partition is equivalent to a coloring of the complement. Since graph coloring is NP-Complete, they used a heuristic.

In the 1980s, high-level synthesis did not attempt to compile complete applications, or even complete procedures. The typical datapath being compiled was specified as a Data Flow Graph (DFG), which represents an acyclic computation. The DFG model can handle conditional branches via if-conversion, but not loops (unless they are unrolled). In 1987, Kurdi and Parker [15] proved that the interference graph for a DFG that had been scheduled belonged to the class of interval graphs, a subclass of chordal graphs. Interval graphs can be colored in $O(V)$ time [24].

Stok [23] studied register allocation for cyclic DFGs, which represent a single loop, and proved that this version of the problem was equivalent to circular arc coloring. Computing the chromatic number of a circular arc graph is NP-Complete. Springer and Thomas [22] studied register allocation for procedures. They found that chordal interference graphs could be guaranteed if certain restrictions are placed on procedure calls and on live ranges in a program. Regular programs, it turns out, effectively have the same restrictions.

2.2 Other Applications of Regular Programs
Many SSA-based optimizations have been published—far too many to enumerate here. This section focuses on algorithms that are specific to regular programs.

Budimlić et al. [4] introduced a fast and effective copy coalescing algorithm that attempts to eliminate as many copies as possible while translating out of SSA Form. By applying Theorem 1, their technique can reason about interferences without constructing an interference graph. Their work was in a Java compiler, so all procedures are regular programs.

Two SSA-based type safety systems [16-17] and one software verification algorithm [14] have been developed that rely on Theorem 1 for their correctness. For these

Using the techniques described in this paper, the algorithms described above could seamlessly be applied to procedures written in C/C++.

3. PRELIMINARIES
Here, we briefly review the data structures and intermediate representations that are used in this paper. Initially, a program is represented as a Control Flow Graph (CFG), which is then converted to SSA Form. During the conversion, a Dominator Tree is used to guide the placement of $\phi$-functions. We require no additional intermediate representations than these three.

3.1 The Control Flow Graph
A CFG is a directed graph $G = (N, E, r)$. $N$ is a set of nodes representing basic blocks, $E$ is a set of edges representing transfers of control flow from one basic block to another, and $r$ is a unique entry node of the CFG. Basic blocks are maximal-length sequences of instructions, with no branches or branch targets interleaved. The notation $m \rightarrow n$ indicates that there is a path from $m$ to $n$ in the CFG. If there is no path, then we write $\neg m \rightarrow n$. By convention, $n \rightarrow n$ if and only if $n$ is contained within a loop.

3.2 The Dominator Tree
In a CFG, node $n$ dominates node $m$ (denoted $n \text{ dom } m$) if and only if every path through the CFG from $r$ to $m$ passes through $n$. If $n$ does not dominate $m$, then we write $\neg n \text{ dom } m$. Trivially, $n \text{ dom } n$ for every basic block $n$; additionally, $r$ dominates every node in the CFG (including itself). Node $n$ strictly dominates node $m$ (denoted $n \text{ sdom } m$) if $n \text{ dom } m$ and $n \neq m$. The immediate dominator of $m$ (denoted $\text{idom } m$) is a node $n$ such that $n \text{ sdom } m$ and there is no other basic block $n'$ such that $n \text{ sdom } n' \text{ sdom } m$; $r$ has no immediate dominator. A dominator tree $T = (N, E_T, r)$ rooted at $r$. Specifically, $E_T = \{(\text{idom } m, m) \mid m \in N-\{r\}\}$. In other words, $T$ contains an edge that connects each node in the CFG to its immediate dominator. A dominator tree can be constructed in $O(|N| + |E|)$ time using the recent algorithm of Georgiadis and Tarjan [9]. We compute the transitive closure of $T$ in $O(|N|^2)$ time and save the result in a bit matrix so that the dominance relation between any pair of CFG nodes can be looked up in $O(1)$ time.

3.3 Static Single Assignment Form
Here, we provide a brief synopsis of SSA form [2, 5-7]. A procedure is in SSA form if (1) every variable is defined once and only if every path through the CFG from $r$ to $m$ passes through $n$. If $n$ does not dominate $m$, then we write $\neg n \text{ dom } m$. Trivially, $n \text{ dom } n$ for every basic block $n$; additionally, $r$ dominates every node in the CFG (including itself). Node $n$ strictly dominates node $m$ (denoted $n \text{ sdom } m$) if $n \text{ dom } m$ and $n \neq m$. The immediate dominator of $m$ (denoted $\text{idom } m$) is a node $n$ such that $n \text{ sdom } m$ and there is no other basic block $n'$ such that $n \text{ sdom } n' \text{ sdom } m$; $r$ has no immediate dominator. A dominator tree $T = (N, E_T, r)$ rooted at $r$. Specifically, $E_T = \{(\text{idom } m, m) \mid m \in N-\{r\}\}$. In other words, $T$ contains an edge that connects each node in the CFG to its immediate dominator. A dominator tree can be constructed in $O(|N| + |E|)$ time using the recent algorithm of Georgiadis and Tarjan [9]. We compute the transitive closure of $T$ in $O(|N|^2)$ time and save the result in a bit matrix so that the dominance relation between any pair of CFG nodes can be looked up in $O(1)$ time.

Two SSA-based type safety systems [16-17] and one software verification algorithm [14] have been developed that rely on Theorem 1 for their correctness. For these
To cope with this conundrum, a \( \phi \)-function \( x_j \leftarrow \phi(x_{j_i}, x_{j_2}) \) is introduced at the join point, and the use of \( x \) is replaced with a use of \( x_j \) in Fig. 1 (b). The semantics of the \( \phi \)-function are as follows: if the path where \( x_j \) is defined is taken, then \( x_j \) receives its value from \( x_{j_i} \); likewise, if the path where \( x_{j_2} \) is defined is taken, then \( x_j \) receives its value from \( x_{j_2} \). In terms of semantics, a \( \phi \)-function is similar in principle to a multiplexer, a simple combinational circuit from logic design. In high-level synthesis, where programs are translated directly to hardware, \( \phi \)-functions are implemented as multiplexers.

Finally, consider a \( \phi \)-function \( y \leftarrow \phi(\ldots x \ldots) \), where \( n \) is the basic block containing the \( \phi \)-function, and \( m \) is the predecessor block which provides variable \( x \) as a parameter of the \( \phi \)-function. This \( \phi \)-function represents a use of \( x \) in \( m \), the predecessor block. The reason is that during SSA deconstruction (as described by Cytron et al. [6] and Briggs et al. [2]), a copy \( y \leftarrow x \) will be appended to \( m \) (the lost copy problem notwithstanding). Consequently, any traversal of \( m \), to determine the variables used within \( m \), must also include a traversal of the \( \phi \)-functions in each successor of \( m \).

### 3.4 Example

Fig. 2 shows a short application written in C/C++ that is an example of a non-strict program. Variable \( x \) is declared before a condition. If the condition is taken, the program exits; however, the compiler simply views the statement `exit(0)` as a typical function call. The compiler is unaware that this function will cause the program to terminate by calling the operating system. If the condition is false, then variable \( x \) is set to value 3. After the condition, \( x \) is output to `stdout` and the program exits normally.

The application uses two variables, `argc` which is passed to the program from the operating system, and \( x \), which is local. Both of these variables are defined exactly once. `argc` is defined as a parameter, and \( x \) is defined within the else clause of the conditional statement. The definition of `argc` dominates its use; the definition of \( x \), however, does not.

Although not at the assembly level, the application in Fig. 2 is in SSA Form, since both \( x \) and `argc` are defined once. However, it is not regular. In the general case, irregular procedures cannot guarantee chordal interference graphs. In this case, the interference graph is chordal, because there are only 2 variables—at least 4 variables are required to generate a non-chordal interference graph.

### 4. NON-STRICK VARIABLE DETECTION

Given a CFG in SSA form, we must first determine whether or not the procedure is strict. At the same time, we must determine which variables violate the strictness of the program. Consider a variable \( v \) that is defined in block \( n \) (or not at all) and used in block \( m \). The use of \( v \) is defined to be:

\[
\text{int main(int argc, char *argv) \{ }
\text{int x;}
\text{if(argc <= 2) \{ exit(0); \}}
\text{else \{ x = 3; \}}
\text{printf("x = %d\n", x);}
\text{return 0;}
\text{\}}
\]

**Figure 2. A non-strict program written in C/C++**

1. `strict` if \( n \) dom \( m \)
2. `non-strict` if \( n \rightarrow m \) and \( \neg n \) dom \( m \)
3. `non-initialized` if \( \neg n \rightarrow m \)
4. `undefined` if \( v \) is not defined anywhere in the program

To fix an undefined use of \( v \), we replace \( v \) with `NULL`, indicating that the use is guaranteed to receive a non-initialized value. If all uses of all variables are strict or undefined, then we are trivially done. Otherwise, we have to fix the SSA representation, which is the topic of the remainder of the paper.

In general, non-initialized uses can be handled the same way as undefined uses. However, we cannot determine whether a use of a variable is non-strict or non-initialized unless we can determine whether or not there is a path from the definition point of the variable to its use. In Case (3) above, \( \neg n \) dom \( m \) is trivially satisfied, so it is not listed. To distinguish between non-strict and non-initialized variables would require computing the transitive closure of the CFG, which has a time complexity of \( O(|N|^2 + |E|) \).

Let \( P \) be the procedure we are currently processing in SSA form. \( \text{NonSI}(P) \) is the subset of variables that have at least one non-strict or non-initialized use. \( \text{NonSIUses}[v] \) is the set of non-strict and/or non-initialized uses of variable \( v \), the first of which will trigger the inclusion of \( v \) into \( \text{NonSI}(P) \). To compute \( \text{NonSI}(P) \) and \( \text{NonSIUses}[v] \), we make two passes over the program.

The first pass computes the basic block where each variable \( v \) is defined, which we denote \( \text{Def}[v] \). If \( v \) is undefined, then \( \text{Def}[v] = \text{NULL} \). The second pass examines each use of each variable \( v \) in the program. If \( v \) is undefined, then each use of \( v \) is replaced with \( \text{NULL} \). If \( v \) is defined somewhere, let \( m \) be the basic block containing the use of \( v \). Then \( v \) is added to \( \text{NonSI}(P) \) if either:

\[
\begin{align*}
(1) \quad & \text{Def}[v] = m, \text{ and the use of } v \text{ precedes the definition of } v \text{ in } m, \text{ or} \\
(2) \quad & \neg \text{Def}[v] \text{ dom } m
\end{align*}
\]

Likewise, the instruction (or \( \phi \)-function) that uses \( v \) is added to \( \text{NonSIUses}[v] \).

In case (1) above, it might be tempting to immediately classify the use of \( v \) that precedes the definition in block \( m \) as a non-initialized use; however, if \( m \) is contained within a loop, the use will be non-initialized on the first execution of the loop. If \( \text{NonSI}(P) \) is empty, then \( P \) is provably a regular program once all undefined variables have been processed accordingly. Otherwise, the variables in \( \text{NonSI}(P) \) must be processed, as described in Sections 4 and 5.

**Theorem 4.** The time complexity of computing \( \text{NonSI}(P) \) and \( \text{NonSIUses}[v] \) for each variable \( v \) is \( O(|N|^2 + |I|) \).
Proof. Let \( V \) be the set of variables in the program, and \( I \) be the set of instructions (including \( \varphi \)-functions). Then the time complexity of computing \( \text{NonSI}(P) \) and \( \text{NonSIUses}[v] \) is \( O(|N|^2 + |V| + |I|) \). The \( O(|V| + |I|) \) term arises from the cost of building the transitive closure of the dominator tree, which enables a constant-time lookup of \(-\text{Def}[v]\ dom m\). The \( O(|N|^2) \) term comes from the cost of initializing \( \text{Def}[v] \) for each variable \( v \in V \), as well as the traversal of all instructions in the program, which includes the definition of each variable in the worst case.

If we assume that each instruction uses at most 2 variables and defines at most 1 variable, then \(|V| \leq 3|I| = O(|I|)\). This simplifies the time complexity to \( O(|N|^2 + |I|) \). □

5. MAKING ONE VARIABLE STRICT

In this section, we outline the theoretical underpinnings that will allow us to translate a program in irregular SSA form into a regular program. To simplify our discussion, we consider a single variable \( v \in \text{NonSI}(P) \) with exactly one use in \( \text{NonSIUses}[v] \). Let \( n \) be the basic block in which \( v \) is defined, and let \( m \) be the basic block containing the use of \( v \). Let \( \text{Reaches}[n] \) be the set of CFG node reachable from \( n \), i.e. \( \text{Reaches}[n] = \{ b \in N | n \rightarrow b \} \). Recall that \( n \in \text{Reaches}[n] \) if and only if \( n \) is contained within a loop. \( \text{Reaches}[n] \) is computed by initiating a depth-first-search from \( n \).

Corollary 2. A use of \( v \) in \( m \) is non-initialized if and only if \( m \notin \text{Reaches}[n] \) and non-strict if and only if \( m \in \text{Reaches}[n] \).

Corollary 2 is illustrated by Figs. 3 (a) and (b); the proof is trivial and has been omitted.

For a non-initialized use, \( v \) is replaced by \( \text{NULL} \). If a compiler or synthesis tool wishes to allocate a register to \( v \), then \( \text{NULL} \) can be replaced with any free register; otherwise, the compiler may wish to display a warning to the user. The warning should display that there is an operation that uses a variable that is not initialized along some path in the application, and that a logical error could result if the program is not rewritten.

To fix a non-strict use of \( v \), a \( \varphi \)-function must be inserted into some block \( m' \). The purpose of the \( \varphi \)-function is to merge the value of \( v \) which flows from predecessors reachable from \( n \), with control flow paths along which \( v \) is not defined. The \( \varphi \)-function in \( m' \) will define a new variable, \( v' \) and the use of \( v \) in block \( m \) will be replaced with a use of \( v' \).

5.1 Selecting the Block for the \( \varphi \)-Function

Here, we prove the correctness of the technique discussed in the prior section. The following section describes how to handle the parameters of the \( \varphi \)-function that is inserted here.

Theorem 5. The correct block to place the \( \varphi \)-function is the block \( m' \) such that (1) \( m' \ dom m \); (2) \( m' \in \text{Reaches}[n] \); and (3) \( m' \) satisfies \( \text{idom} \ m' \notin \text{Reaches}[n] \).

Proof. First, note that block \( m' \) must exist since \( r \notin \text{Reaches}[n] \) and \( m \ dom m \). The three criteria are addressed as follows. Contradiction is used in each case.

(1) Let \(-m' \ dom m \). Then we have replaced a non-strict use with another non-strict use, and have fixed nothing.

(2) Let \( m' \notin \text{Reaches}[n] \). Then for every predecessor block \( m'' \) of \( m \), \( m'' \notin \text{Reaches}[n] \). Hence, every parameter of the \( \varphi \)-function is \( \text{NULL} \). Therefore \( v' \) is guaranteed to be \( \text{NULL} \) when it is defined, and the use of \( v' \) in block \( m \) is replaced with a use of \( \text{NULL} \) instead. We have converted a non-strict use to a non-initialized use, altering the semantics of the application being compiled.

(3) Let \( \text{idom} \ m' \in \text{Reaches}[n] \). For each predecessor \( m'' \) of \( m' \), \( m'' \in \text{Reaches}[n] \). Therefore \( v \) will be the parameter of every slot in the \( \varphi \)-function. First and foremost, a \( \varphi \)-function that has the same parameter value in every slot is an identity operation that does not change the state of the system, regardless of which path is taken into \( m' \). Since \( v \) is now used at the end of every predecessor of \( m' \), it suffices to treat this \( \varphi \)-function as a “pseudo-use” of \( v \) in \( m' \).

There is a path from \( r \) to \( m' \) that does not go through \( n \); otherwise, \( n \ dom m \), contradicting the assumption that the use of \( v \) at \( m \) is non-strict. The “pseudo-use” of \( v \) in \( m' \) is non-strict as well.

Let us suppose that an oracle inserts a new \( \varphi \)-function that defines a variable \( v' \) that “fixes” the pseudo-use of \( v \) at \( m' \). Therefore the “pseudo-use” of \( v \) can be replaced with a “pseudo-use” of \( v' \). Let \( B \) be the block containing the new \( \varphi \)-function. By (1) above, \( B \ dom m' \ dom m \). Hence, the use of \( v \) at \( m \) can be fixed by a use of \( v' \). In short, the \( \varphi \)-function inserted to fix the pseudo-use of \( v \) at \( m' \) effectively fixed the non-strict use of \( v \) at \( m' \) as well. Hence, the \( \varphi \)-function inserted at \( m' \) fixed nothing. □

Fig. 4 illustrates the main points of case (3) in Theorem 5. Fig. 4 (a) shows what would occur if \( \text{idom} \ m' \in \text{Reaches}[n] \) — all of the parameters of the \( \varphi \)-function inserted into \( m' \) would be non-strict uses—and none of them would be \( \text{NULL} \). Fig. 4 (b) shows a new block \( B \) that contains a \( \varphi \)-function inserted by an oracle to fix the non-strict uses of \( v \) in the \( \varphi \)-function in \( m' \). Fig. 4 (c) shows that since \( B \ dom m' \ dom m \), it suffices to eliminate the \( \varphi \)-function in \( m' \) and replace the non-strict use of \( v \) in \( m' \) with \( v'' \), the variable defined by the \( \varphi \)-function in \( B \). Fig. 4 (d) shows the correct solution, as stated by Theorem 5. The \( \varphi \)-function should be placed in the node \( m' \) such that such that \( m' \ dom m, m' \in \text{Reaches}[n], \) and \( \text{idom} \ m' \notin \text{Reaches}[n] \). Node \( B \) must satisfy these same conditions for both \( m \) and \( m' \) in Fig. 4 (b) and (c). Since \( \text{idom} \ m' \notin \text{Reaches}[n] \) in Fig. 4 (d), then \( n \in \text{Reaches} [\text{idom} \ m' \) ; otherwise, the use of \( v \) in node \( m \) would be non-initialized.
Handling the $\phi$-Function Parameters

By applying Theorem 5, we know how to insert a $\phi$-function that fixes a non-strict use of a variable. Moreover, the proposed technique ensures that at least one parameter slot of the $\phi$-function must be NULL—this is, of course, necessary since the variable $v$ in question has not been defined along at least one path leading to its use; and the $\phi$-function is placed at a place where a path along which $v$ has been defined converges with a path from $r$ along which $v$ has not been defined. There is no guarantee, however, that all (or any non-NULL) parameters of the $\phi$-function will be either strict or non-strict uses. This section addresses the new uses that are created by inserting the $\phi$-function. Let $m'$ be the basic block where the new $\phi$-function was inserted, and let $m''$ be a predecessor of $m$ that supplies a parameter to the $\phi$-function.

There are 3 cases to consider:

1. If $n \text{ dom } m''$, then the $\phi$-function parameter is a strict use of $v$. This subsumes the case $n = m''$, since the use is appended to the end of the block.

2. If $\neg n \text{ dom } m''$ and $m'' \notin \text{ Reaches}[n]$, then the $\phi$-function parameter is a non-initialized use of $v$; the parameter is replaced by NULL.

3. If $\neg n \text{ dom } m''$ and $m'' \in \text{ Reaches}[n]$, then the $\phi$-function parameter is a non-strict use of $v$. The $\phi$-function parameter is added to NonSIUses[$v$], and the process described in Section 5.1 repeats, with $m'' = m$.

Cases (2) and (1)/(3) are illustrated by Figs. 5 (a) and (b) respectively. Fig. 5 (b) is ambiguous about whether or not $n \text{ dom } m''$ in order to handle both cases.

5.2 Handling the $\phi$-Function Parameters

By applying Theorem 5, we know how to insert a $\phi$-function that fixes a non-strict use of a variable. Moreover, the proposed technique ensures that at least one parameter slot of the $\phi$-function must be NULL—this is, of course, necessary since the variable $v$ in question has not been defined along at least one path leading to its use; and the $\phi$-function is placed at a place where a path along which $v$ has been defined converges with a path from $r$ along which $v$ has not been defined. There is no guarantee, however, that all (or any non-NULL) parameters of the $\phi$-function will be either strict or non-strict uses. This section addresses the new uses that are created by inserting the $\phi$-function. Let $m'$ be the basic block where the new $\phi$-function was inserted, and let $m''$ be a predecessor of $m$ that supplies a parameter to the $\phi$-function.

There are 3 cases to consider:

1. If $n \text{ dom } m''$, then the $\phi$-function parameter is a strict use of $v$. This subsumes the case $n = m''$, since the use is appended to the end of the block.

2. If $\neg n \text{ dom } m''$ and $m'' \notin \text{ Reaches}[n]$, then the $\phi$-function parameter is a non-initialized use of $v$; the parameter is replaced by NULL.

3. If $\neg n \text{ dom } m''$ and $m'' \in \text{ Reaches}[n]$, then the $\phi$-function parameter is a non-strict use of $v$. The $\phi$-function parameter is added to NonSIUses[$v$], and the process described in Section 5.1 repeats, with $m'' = m$.

Cases (2) and (1)/(3) are illustrated by Figs. 5 (a) and (b) respectively. Fig. 5 (b) is ambiguous about whether or not $n \text{ dom } m''$ in order to handle both cases.

Figure 4.
Illustration of Case (3) of Theorem 5. The (useless) $\phi$-function inserted in $m'$ (a); a second $\phi$-function inserted into block B by an oracle that fixes all of the non-strict uses of $v$ as parameter of the first $\phi$-function (b); the second $\phi$-function would effectively fix the non-strict use of $v$ in $m$ if we get rid of the first second $\phi$-function (c); and the correct solution (d).

Figure 5. Illustration of Cases (2) (a) and (1)/(3) (b) respectively.

In Fig. 5, Straight arrows transfer control from block $m''$ to successor $m'$. Curved arrows are paths in the CFG. Dashed arrows represent the flow of data values that converge at the $\phi$-function in $m'$.

Theorem 6. The three cases above correctly handle all parameters of the $\phi$-function that has been inserted into $m'$.

Proof. Cases (1) and (2) are trivial, since we are satisfied with both strict and non-initialized uses of $\phi$-function parameters.

With respect to Case (3), we have a non-strict use. By Theorem 5, we know how to insert a $\phi$-function to fix a non-strict use. To see that the algorithm eventually converges, observe that when a $\phi$-function is placed in node $m'$ that defines a new variable $v'$,
replacing each non-strict use of $v$ in the sub-tree of the dominator tree rooted at $m'$ ensures that the resulting use is strict.

Now, note that $\phi$-placement algorithm described in Section 5.1 and proven correct by Theorem 5 always moves up the dominator tree toward the root, $r$. All nodes eventually will be processed and the algorithm terminates.  

6. MAKING ALL VARIABLES STRICT

Here, we discuss how to impose strictness of all uses of a variable by making two passes over the dominator tree. The first pass inserts $\phi$-functions at the necessary places to take care of non-strict variables. The second pass renames all of the non-strict/non-initialized uses of each variable. In practice, the initial pass will be performed for each variable in $NonSI(P)$.

The second pass executes one time, concurrently renaming each use of each variable in $NonSI(P)$. This is faster than performing the second pass repeatedly for each non-strict variable.

Consider variable $v \in NonSI(P)$ and let $m$ be a block that contains a non-strict use of $v$. Starting with any non-strict use in $NonSIUses[v]$, a $\phi$-function is inserted in block $m'$ as described Section 5. All blocks dominated by $m'$ are marked and will not be processed again for $v$. All uses of $v$ in marked blocks can be ignored because they can be fixed by renaming during the second pass. This process repeats until there are no variables left in $NonSIUses[v]$. Clearly, at most $O(|N|)$ blocks will ever be marked, since no block is marked twice. The marking phase of this pass is essentially a post-order traversal of the sub-tree of the dominator tree rooted at $m'$, which ensures termination.

The second pass traverses the dominator tree in reverse post-order and renames each use of variable. This pass is similar in principle to the renaming phase of SSA construction [2]; the differences are that no $\phi$-functions are inserted during this pass and only variables in $NonSI(P)$ are ever renamed.

$NameStack[...]$ is an array of stacks, one for each variable $v \in NonSI(P)$. At the root, NULL is pushed onto $NameStack[v]$ for each variable $v$. When the definition of $v$ is encountered, $v$ is pushed onto $NameStack[v]$. When $a$

At the root node, NULL is pushed onto the stack for each variable. When the instruction that defines $v$ is found, at each $\phi$-function inserted for $v$, we push a new name $v_{index[v]}$ onto the stack. Initially, $index[v]$ is set to 1 and is incremented each time a new variable $v$ is found. When a use of variable $v$ is found—

including uses as $\phi$-function parameters, which are associated with the predecessors of the block containing the $\phi$-function—the use is replaced with the element on top of $NameStack[v]$.

The reverse post-order traversal is implemented recursively. After processing each CFG node $n$, all of the variables $v$ such that a new name has been pushed onto $NameStack[v]$ are saved in a separate stack; this is used so that the $NameStack[...]$ can be restored to its original state before $n$ was processed. After saving the state, each child of $n$ in the dominator tree is processed recursively. By having each child restore the stack at the end of the recursion, $NameStack[...]$, each recursive call to a child begins with identical values in $NameStack[...]$.

**Theorem 7.** The algorithm described above correctly translates an irregular program in SSA Form into a regular program.

**Proof.** Assume to the contrary that after running the algorithm, there exists a non-strict or non-initialized use of variable $v$.

If the use is non-initialized, then $NameStack[v].top$ will be equal to NULL when the basic block containing the use is processed. Therefore the use of $v$ will be overwritten with a use of NULL, contradicting the assumption.

If the use is non-strict, then by Theorem 6, a $\phi$-function that defines a new variable for $v$ will be inserted into a basic block that dominates the non-strict use of $v$. Therefore, the renaming phase must have failed; however, the block containing the $\phi$-function is processed before the block containing the non-strict use of $v$ since the dominator tree is traversed in reversed post-order. If $v_1$ is the name given to the variable defined by the $\phi$-function, then $v_1$ is on top of $NameStack[v]$ when the non-strict use of $v$ is encountered.

Thus, the non-strict use of $v$ is replaced with a strict use of $v$, a contradiction.  

6.1 Time Complexity

Let $G = (N, E)$ be the CFG for procedure $P$ and $|I|$ be the total number of instructions in the CFG, including the set of $\phi$-functions that are there originally. Let $V$ be the set of variables in the program, and $x = |NonStrict(P)|$ be the number of variables having non-strict uses. If we assume that the CFG does not contain multi-way branches, then each CFG node has at most 2 successors, so $E = O(|N|)$.

**Theorem 8.** The time complexity of converting from an irregular SSA Form program to a regular program is $O(|N|^2 + x|N| + |I|)$.

**Proof.** There are several different ways that $Reaches[n]$ could be computed. First and foremost, a depth-first search could be initiated from every basic block $n$ containing the definition of a non-strict variable. The time complexity is $O(|N| + |E|) = O(|x|N)$ in this case. Alternatively the transitive closure could be computed in $O(|N|E)$ time. Since $x$ is usually quite small, we adopt the first approach. In the worst case, $x = |I|$, if we assume that each instruction defines at most one variable.

The time complexity of inserting $\phi$-functions is $O(|X|N) + |I|). The $O(x|N)$ is due to the fact that the first pass over the dominator tree is processed once for each variable in $NonStrict(P)$, and the fact that $O(|N|)$ nodes in the dominator tree are marked per pass. Additionally, at most $x|N|$ $\phi$-functions will be inserted during this pass—one $\phi$-function for each variable in $NonSI(P)$, inserted into each basic block. The $O(|I|)$ term arises do to the fact that each non-strict or non-initialized use must be removed from $NonStrictUses[v]$ at some point. In the absolute worst case, every instruction in the program that uses $v$ will be placed in $NonStrictUses[v]$.

The complexity of the second traversal of the dominator tree is $O(|N|x + |I|)$. Each CFG node is processed once during the traversal, yielding a time complexity of $O(|N|)$. Each instruction in the program is traversed as well at a cost of $O(|I|)$.

Additionally, the extra $\phi$-functions are processed, at a cost of $O(x|N)$. Note that no more than $x|N|$ $\phi$-functions can be inserted—one for each non-strict variable inserted into every basic block; this also ensures that there are at most $O(E) = O(x|N)$ $\phi$-function parameters inserted into the application.
From Theorem 4, the time complexity of computing NonSI(P) and NonSIUses[v] for each variable v is \(O(|N|^2 + |I|)\). Combining this with the terms derived previously, the overall time complexity is \(O(|N|^2 + x|N| + |I|)\). □

7. EXPERIMENTAL RESULTS

We have implemented the techniques described in the preceding sections into the Machine SUIF compiler [20], using pruned SSA form [5]. Prior to applying this transformation, we ran one round of SSA-based dead code elimination [18, p. 219-222] to remove unreachable basic blocks, which may exist in the intermediate representation. We compiled the C-language benchmark suites from MiBench [10] and SPEC INT 2000.

The results for all of the non-strict functions that we found are summarized in Tables 1 and 2 respectively. Results are only shown for benchmarks that exhibited at least one non-strict function. The Rijndael and 181.crafty benchmarks were not compiled due to assertion failures that occurred in Machine SUIF’s code generation. 252.eon was not compiled because it is written in C++, and Machine SUIF’s front end only compiles C-language programs. The columns labeled Non-strict Functions and Non-strict Variables list the number of non-strict functions and variables found in each benchmark respectively. The column labeled Phis Inserted lists the number of phi-functions inserted for each program. The column labeled Variable Uses Renamed lists the number of uses of each variable (both instruction uses and phi-function parameters) that were renamed.

Following the transformation, the procedure outlined in 4 was run to ensure the correctness of the resulting transformation. Every irregular function was translated into regular form. Clearly, 176.gcc, 253.perlmbk, and ghostscript had considerably more non-strict functions and variables than the others.

Of then 19 benchmarks listed in Tables 1 and 2, 12 required only 1 or 2 phi-functions in order to ensure regular form.

Tables 3 and 4 compare the runtime of converting to a regular program to the runtime of building pruned SSA form. Column Make strict lists the runtime (microseconds) of converting from irregular SSA Form to a regular program. The column Pruned lists the runtime of converting pruned SSA form. The column labeled Overhead lists the fraction of time enforcing strictness relative to the total time required to build pruned SSA form.

If \(T_{prune}\) is the time to build pruned SSA form, and \(T_{prune}\) is the time to impose strictness on the program using the techniques described in this paper, then Overhead = \(T_{prune} / (T_{prune} + T_{prune})\), expressed as a percent. From the Overhead columns Tables 3 and 4, the cost of Make strict relative to Pruned ranges from 6.14% (176.gcc) to 15.89% (gsm). For the 3 benchmarks that required the largest number of phi-functions, the overhead was 6.14% (176.gcc), 7.03% (253.perlmbk), and 8.97% (ghostscript).

Machine SUIF’s SSA library places phi-functions using the dominance frontier algorithm of Cytron et al. [5]. To the best of our knowledge, the fastest phi-placement algorithm is due to Das and Ramakrishna [7], who report speedups of 17% for SPEC INT 2000 and 37% for SPEC FP 2000. The speedup is reported for the actual placement of phi-functions and does not appear to include the cost of performing liveness analysis; moreover, they don’t explicitly state which flavor of SSA form is used.

### Table 1. Results of conversion to regular SSA form for SPEC INT 2000 benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Non-strict Functions</th>
<th>Non-strict Variables</th>
<th>Phis Inserted</th>
<th>Uses Renamed</th>
</tr>
</thead>
<tbody>
<tr>
<td>175.vpr</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>176.gcc</td>
<td>100</td>
<td>170</td>
<td>173</td>
<td>756</td>
</tr>
<tr>
<td>197.parser</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>253.perlmbk</td>
<td>80</td>
<td>101</td>
<td>101</td>
<td>276</td>
</tr>
<tr>
<td>254.gap</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>300.twolf</td>
<td>24</td>
<td>33</td>
<td>33</td>
<td>127</td>
</tr>
</tbody>
</table>

### Table 2. Results of conversion to regular SSA form for MiBench benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Non-strict Functions</th>
<th>Non-strict Variables</th>
<th>Phis Inserted</th>
<th>Uses Renamed</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>bitcount</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>ghostscript</td>
<td>78</td>
<td>116</td>
<td>122</td>
<td>401</td>
</tr>
<tr>
<td>gsm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ispell</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>jpeg6a</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td>56</td>
</tr>
<tr>
<td>ppg</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>sphinx</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>stringsearch</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>susan</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>tiff</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>typeset</td>
<td>26</td>
<td>64</td>
<td>66</td>
<td>629</td>
</tr>
</tbody>
</table>

### Table 3. Runtime of conversion to strict SSA form and building pruned SSA form for SPEC INT 2000

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Make_strict (usec)</th>
<th>Pruned (usec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>175.vpr</td>
<td>1680</td>
<td>8544</td>
<td>16.43%</td>
</tr>
<tr>
<td>176.gcc</td>
<td>2820623</td>
<td>43140771</td>
<td>51.14%</td>
</tr>
<tr>
<td>197.parser</td>
<td>5288</td>
<td>64194</td>
<td>10.27%</td>
</tr>
<tr>
<td>253.perlmbk</td>
<td>609243</td>
<td>8066809</td>
<td>7.03%</td>
</tr>
<tr>
<td>254.gap</td>
<td>10428</td>
<td>93613</td>
<td>10.02%</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>14904</td>
<td>128242</td>
<td>10.41%</td>
</tr>
<tr>
<td>300.twolf</td>
<td>354618</td>
<td>3904853</td>
<td>8.33%</td>
</tr>
</tbody>
</table>

### Table 4. Runtime of conversion to strict SSA form and building pruned SSA form for MiBench benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Make_strict (usec)</th>
<th>Pruned (usec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>3446</td>
<td>38251</td>
<td>8.26%</td>
</tr>
<tr>
<td>bitcount</td>
<td>800</td>
<td>6771</td>
<td>10.57%</td>
</tr>
<tr>
<td>ghostscript</td>
<td>735149</td>
<td>7456596</td>
<td>9.97%</td>
</tr>
<tr>
<td>gsm</td>
<td>951</td>
<td>5035</td>
<td>15.89%</td>
</tr>
<tr>
<td>ispell</td>
<td>6072</td>
<td>46976</td>
<td>11.45%</td>
</tr>
<tr>
<td>jpeg6a</td>
<td>12802</td>
<td>103604</td>
<td>11.00%</td>
</tr>
<tr>
<td>ppg</td>
<td>63162</td>
<td>737667</td>
<td>7.89%</td>
</tr>
<tr>
<td>sphinx</td>
<td>13843</td>
<td>81936</td>
<td>14.45%</td>
</tr>
<tr>
<td>stringsearch</td>
<td>1686</td>
<td>10861</td>
<td>13.44%</td>
</tr>
<tr>
<td>susan</td>
<td>12934</td>
<td>140848</td>
<td>8.41%</td>
</tr>
<tr>
<td>tiff</td>
<td>19997</td>
<td>125614</td>
<td>13.73%</td>
</tr>
<tr>
<td>typeset</td>
<td>1349875</td>
<td>13901981</td>
<td>8.85%</td>
</tr>
</tbody>
</table>
If we conservatively apply a 17% reduction in the runtime of building pruned SSA form to the results in Table 4, the aggregate overhead jumps from 6.45% to 7.67%; translating to regular SSA form is still much faster than rebuilding SSA form.

8. CONCLUSION

A technique to transform a non-strict SSA Form program into a regular program has been presented. This technique ensures that register allocation in high-level synthesis can be solved optimally in polynomial-time for any procedure in SSA Form, regardless of whether or not the procedure is strict. As shown in the experiments, the conversion algorithm is faster than rebuilding SSA Form from scratch.

REFERENCES