ABSTRACT
Static Single Information (SSI) Form is a compiler intermediate representation that extends the more well-known Static Single Assignment (SSA) Form. In 2005, several research groups independently proved that interference graphs for procedures represented in SSA Form are chordal graphs. This paper performs a similar analysis concerning SSI Form, and proves that interference graphs are interval graphs. The primary consequences of this paper are threefold: (1) Linear scan register allocation for programs in SSA Form can be implemented in such a way that there are no lifetime holes, thereby sidestepping one of the drawbacks that plagued non-SSI implementations; (2) the $k$-colorable subgraph problem can be solved in polynomial-time for interval graphs, but remains NP-Complete for chordal graphs—to date, no register allocation algorithms have been implemented that solve the $k$-colorable subgraph problem directly; and (3) liveness analysis converges after a single iteration for programs represented in SSI Form.

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

General Terms
Algorithms, Performance, Languages

Keywords
Static Single Information (SSI) Form, Compilers, Register Allocation, Linear Scan Register Allocation, Interval Graph, $k$-Colorable Subgraph Problem

1. INTRODUCTION
Register allocation is probably the most widely studied back-end optimization in compiler theory. Numerous heuristics have been proposed to solve the problem, as well as optimal algorithms that run in worst-case exponential time. The vast majority of register allocation algorithms use a Control Flow Graph (CFG) as the main data structure to represent a program. There are exceptions, such as the linear scan family of algorithms [21, 23, 26, 28, 30] that flatten the CFG into a linear list of instructions, and an older algorithm described by Norris and Pollack [22] that uses the Program Dependence Graph. Nonetheless, most algorithms that have been implemented in real-world compilers either use the CFG or implement some variation of linear scan.

Recently, several research groups have independently studied register allocation for programs represented in Static Single Assignment (SSA) Form. As discussed in Section 2, they have proven that interference graphs for programs in SSA Form are chordal graphs [4-5, 17]. Despite this fascinating result, register allocation in compilers remains NP-Complete [4, 10, 24].

Admittedly, the problem does become polynomial in the context of high-level synthesis [5]; however, synthesis is beyond the scope of this work.

This paper, in contrast, explores the possibility of performing register allocation for programs in Static Single Information (SSI) Form rather than SSA Form. In particular, we prove that the interference graph for a program in SSI Form is an interval graph, a subclass of the chordal graphs. The consequences of this result will be contrasted with SSA Form in sections 2.1-2.3.

2. RELATED WORK
Over the last 2 years, several research groups have studied the possibility of performing register allocation on programs represented in SSA Form. The primary advantage of this representation is that interference graphs for SSA form programs belong to the class of chordal graphs. Chordal graphs are an interesting class of graphs, because several problems that are NP-Complete for general graphs have polynomial-time solutions for chordal graphs, including: maximum clique and independent set, minimal coloring and covering by cliques [14], and maximum vertex-weighted independent set [12].

None of these theoretical problems, however, corresponds directly to any of the problems faced by register allocation, namely spilling and coalescing. For example, computing the chromatic number $\chi(G)$ of an interference graph $G$ is useful if $\chi(G) \leq R$, where $R$ is the number of registers in the target architecture; however, if $\chi(G) > R$, a chordal interference graph cannot tell us which variables ought to be spilled. Likewise, computing $\chi(G)$ does not tell us the best assignment of variables to registers in order to minimize the number of copy and swap instructions required to translate a procedure out of SSA form [18, 24]. At present, it is not immediately clear whether SSA-based register allocation will produce a quantifiable improvement over established non-SSA-based techniques.

2.1 The $k$-Colorable Subgraph Problem
The most important difference between interval graphs and chordal graphs has to do with the $k$-colorable subgraph problem, which is a reason, albeit inexact, model for spilling. If $k$ is the number of registers in the target architecture, then the $k$-colorable subgraph problem computes the largest (vertex-weighted) subgraph $G' = (V', E')$ of the interference graph $G = (V, E)$, such that $\chi(G') \leq k$. The variables corresponding to vertices in $V'$ are stored in registers, while those corresponding to vertices in $V - V'$ are stored in memory. The inexactness of this model arises from the fact that variables that are spilled must reside in registers immediately after they are computed and at points in the program where they are used by other instructions.

For chordal graphs, the $k$-colorable subgraph problem is NP-Complete if $k$ is unbounded, and polynomial in $k$ if $k$ is a constant [31]. In other words, this means that the runtime of the problem
will be of the order $O(n^3)$; for a target architecture with 32 registers, for example, this means that the running time will be $O(n^3)$. In this case, polynomial does not automatically mean sufficiently efficient for an implementation in a real-world compiler.

For interval graphs, on the other hand, this problem is polynomial regardless of whether $k$ is unbounded or constant [31], and the algorithm that solves the problem is much more efficient [11]; in fact, similar ideas have been integrated into a register binding and port assignment framework used for high-level synthesis [6]. In short, we believe that it is possible to design and implement a register allocation framework that solves this problem directly for interval graphs.

### 2.2 Linear Scan Register Allocation

A second advantage of using SSI Form involves the linear scan family of register allocation algorithms [21, 23, 26, 28, 30]. Linear scan flattens a CFG into a linear list of instructions before performing register allocation and instructions are numbered in increasing order starting at 1. Linear scan represents each variable $v$ as a lifetime interval $[i, j)$ such that: (1) $i$ is the largest instruction index such that there is no index $i' < i$ such that $v$ is live at $i'$; and (2) $j$ is the smallest instruction index such that there is no index $j' > j$ such that $v$ is live at $j'$. Register allocation then proceeds over the intervals using an adaptation of Belady’s Algorithm [3] for cache page replacement—at each point where a variable must be spilled, spill the variable that currently resides in a register whose next use is furthest away.

Linear scan register allocation is ideal for situations where the runtime of the compiler is more important than the quality of the code that it produces. Often, this occurs in embedded systems in the context of just-in-time compilation. Linear scan is one of the fastest register allocations to be implemented; however, it suffers from one major drawback: there may be lifetime holes, i.e. sub-intervals within $[i, j)$ where $v$ is not actually live. To improve the quality of code generated by linear scan, several authors have augmented linear scan with a second pass that attempts to promote some spilled variables back into registers [21, 28, 30].

Using SSI Form, on the other hand, each interval $[i, j)$ corresponds directly to the lifetime of a variable. In other words, $v$ is live at every point within $[i, j)$ and is not live at any point outside of $[i, j)$. In other words, there are no lifetime holes. Consequently, an implementation of linear scan using SSI Form can reap the traditional benefit of linear scan—fast runtime, but without the main detraction—lifetime holes.

### 2.3 Liveness Analysis

Any instruction of the form $v \leftarrow ...$ is a definition of $v$; any instruction of the form $... \leftarrow v$ is a use of $v$. Variable $v$ is defined to be live at some point $p$ in a program if there is a path from at least one definition of $v$ to $p$, and a path from $p$ to at least one use of $v$. Liveness Analysis is the process of determining the set of variables that are live at each point in the program.

If $n$ is a basic block in a CFG, LIVEOUT($n$) is defined to be the set of variables that are live at the exit point of $n$. Given LIVEOUT($n$), the set of variables that are live at each point within $n$ can easily be determined by traversing the instructions within $n$ in reverse order; this same traversal can also be used to construct an interference graph.

Liveness analysis proceeds using a process called *iterative data flow analysis*; this process is described as follows. First, we must introduce two additional sets for each basic block:

- $UEVAR(n)$ is the set of *upwards-exposed* variables in block $n$—these are variables that are used in block $n$, but have no definitions preceding them. Therefore, they will belong to the LIVEOUT sets of each predecessor of $n$.
- $VARKILL(n)$ is the set of variables that are defined in block $n$, but are not used prior to their definitions in $n$.

For each block $n$, the set LIVEOUT($n$) is initially empty. Then, the following equation updates LIVEOUT($n$).

$$LIVEOUT(n) = LIVEOUT(n) \cup \bigcup_{m \in succ(n)} (UEVAR(m) \cup LIVEOUT(m) \cap \overline{VARKILL(m)})$$

This equation is solved for each basic block $n$—this process is called an *iteration*. If the LIVEOUT set changes for at least one basic block, then another iteration is performed. The process repeats until none of the LIVEOUT sets changes for any of the blocks.

Let $V$ be the set of variables in the program, $N$ be the number of basic blocks, and $I$ be the number of instructions in the program.

To simplify the analysis, assume that $I$ uses and defines a constant number of variables. The worst-case time complexity of liveness analysis is $O(I + VN)$; in practice, the runtime typically lies somewhere between $O(I + VN)$ and $O(I + VN^2)$, and does not approach the asymptotic worst case. Nonetheless, liveness analysis is one of the costliest stages of register allocation in terms of runtime [7].

Using SSI Form, we will show that liveness analysis requires only a single iteration to converge; additionally, the $UEVAR$ and $VARKILL$ sets are not required, thereby reducing the amount of space needed as well. If desired, the interference graph can be constructed during the same traversal. This should have runtime advantages for any register allocation algorithm that uses SSI Form, regardless of whether it belongs to the linear scan or graph-coloring families of algorithms.

### 3. PRELIMINARIES

Here, we present preliminary information that is necessary to understand the contributions of this paper. Section 3.1 introduces SSA Form, and Section 3.2 introduces SSI Form, which is an extension of SSA Form. Section 3.3 introduces the dominance relation, a fundamental concept in compiler theory which will be used to prove that interference graphs for SSI Form programs are interval graphs. Interval graphs will be defined in Section 3.4.

#### 3.1 SSA Form

*Static Single Assignment (SSA)* Form is an intermediate representation for a procedure that was introduced in the late 1980s by a series of papers from IBM [1, 25, 29] in the context of specific compiler optimizations. It did not gain wide acceptance, however, until a 1991 paper by Cytron et al. [9] introduced it to the wider community.
A procedure is defined to be in SSA Form if every variable is defined exactly once, and every use of a variable corresponds to one definition. Of central importance to SSA Form is the concept of a \( \phi \)-function, which is used to merge multiple variables into one at points in the program where different paths through the CFG join together.

Fig. 1 (a) illustrates a CFG fragment, where variable \( v \) has been defined twice and used once. In Fig. 1 (b), both definitions of \( v \) have been renamed to \( v_1 \) and \( v_2 \) respectively. Referring back to Fig. 1 (a), simply renaming \( v_1 \) and \( v_2 \) is not enough—because the use of \( v \) could not be named to either \( v_1 \) or \( v_2 \) and still retain the semantics of the original program. In Fig. 1 (b), a \( \phi \)-function is introduced. Similar in principle to a multiplexer in hardware design, the \( \phi \)-function merges \( v_1 \) and \( v_2 \) into a new variable, \( v_3 \), and replaces the use of \( v \) with a use of \( v_3 \) instead. The semantics of the \( \phi \)-function are as follows: if the left path is taken into the block that uses \( v \), then \( v_3 \) receives the value of \( v \); if the right path is taken, then \( v_3 \) receives the value of \( v_2 \).

\( \phi \)-functions are placed at confluence points, where multiple paths in a CFG merge. A \( \phi \)-functions placed in block \( n \) will have as many parameter slots as \( n \) has predecessors, and it is possible that the same variable will occur in multiple parameter slots.

One important fact about \( \phi \)-functions is noted here: the definition of the variable defined by the \( \phi \)-function is associated with the basic block that contains the \( \phi \)-function. The use of each parameter, on the other hand, is associated with the predecessor block corresponding to each parameter slot. With respect to Fig. 1 (b), the use of \( v_1 \) is associated with the end of the block that defines \( v_2 \); likewise, the use of \( v_2 \) is associated with the end of the block that defines \( v_3 \).

If the uses of the variables by the \( \phi \)-function correspond to the block containing the \( \phi \)-function, then \( v_1 \) and \( v_2 \) would interfere with one another—and in practice, they should not. \( v_1 \) and \( v_2 \) will never be live at the same time since their entire lifetimes occur on mutually exclusive paths in the CFG that happen to converge at the same point.

### 3.2 SSI Form

The Static Single Information (SSI) Form was introduced by Ananian [2] in order to improve the quality of backward dataflow analyses; it was then formalized and extended by Singer [27]. SSI form extends SSA form in the following way. SSA form inserts \( \phi \)-functions at points where divergent control flow paths in a program merge together, and a variable has been defined at multiple points along these separate paths. In addition to \( \phi \)-functions, SSI form inserts \( \sigma \)-functions at split points, such as conditional branches. \( \phi \)-functions effectively merge multiple variables into one unique variable name; \( \sigma \)-functions, in contrast, split one variable into multiple ones—alikeous to a demultiplexer in logic design.

Fig. 2 (a) shows a CFG fragment converted to SSA form in Fig. 2 (b) and SSI Form in Fig. 2 (c). In Fig. 2 (a), variable \( v \) is used on both sides of a condition before it is redefined. A \( \sigma \)-function is inserted at the entry block to split \( v \)—whose initial use has been renamed to \( v_1 \)—into two new variables, \( v_2 \) and \( v_3 \) that are used on both sides of the condition.

The definition of an SSI variable defined by a \( \sigma \)-function is associated with the successor node of the condition—similar in principle to the association of uses of \( \phi \)-function parameters with predecessor blocks. In Fig. 2 (c), the definition of \( v_2 \) is associated with the block on the left-hand-side of the condition, and the definition of \( v_3 \) is associated with the block on the right-hand-side. The remaining variables have semantics derived from the preceding discussion involving SSA form.

### 3.3 Dominance

A Control Flow Graph (CFG) is a directed graph \( G = (N, E, r, t) \) representing a procedure. \( N \) is a set of basic blocks, \( E \) is a set of edges representing control flow transfers from one basic block to another, and \( r \) and \( t \) are distinguished entry and exit nodes. Both \( r \) and \( t \) are empty, i.e. they contain no instructions, \( \phi \)-, or \( \sigma \)-functions. \( r \) executes implicitly when the procedure is entered, and \( t \) executes implicitly when it exits.
Let \( n \) and \( m \) be nodes in the CFG. \( n \) dominates \( m \) (denoted \( n \ dom m \)) if every path from \( r \) to \( m \) passes through \( n \). By convention, every node dominates itself, i.e. \( n \ dom n \) is always true. \( n \) strictly dominates \( m \) (denoted \( n \ sdom m \)) if \( n \ dom m \) and \( n \neq m \). \( n \) post-dominates \( m \) (denoted \( n \ pdom m \)) if every path from \( m \) to \( t \) passes through \( n \). \( n \) strictly post-dominates \( m \) (denoted \( n \ spdom m \)) if \( n \ sdom m \) and \( n \neq m \).

The immediate dominator of \( n \) (denoted \( idom n \)) is a node \( m \) such that \( m \ sdom n \) and there is no other node \( m' \) such that \( m' \ sdom n \). Likewise, the immediate post-dominator of \( n \) (denoted \( ipdom n \)) is a node \( m \) such that \( m \ spdom n \) and there is no other node \( m' \) such that \( m' \ spdom n \).

Every node in the CFG has an immediate dominator, except for \( t \), and every node has an immediate post-dominator, except for \( t \). A dominator tree is an undirected graph \( D = (N, E_D, r) \), where \( r \) is the root of the tree and \( E_D = \{ (n, idom n) \mid n \in N - \{ r \} \} \) is a set of edges that connects each node \( n \) to its immediate dominator. Likewise, a post-dominator tree is an undirected graph \( P = (N, E_P, t) \), where \( t \) is the root and \( E_P = \{ (n, ipdom n) \mid n \in N - \{ t \} \} \) is a set of edges that connects each node \( n \) to its immediate post-dominator.

Dominator information is required to construct both SSA and SSI Form. This information can be computed in \( O(N + E) \) time using the algorithm of Georgiadis and Tarjan [15]; an asymptotically slower implementation by Lengauer and Tarjan [20] tends to run slightly faster in practice; its time complexity of this algorithm is \( O((N + E)\alpha(N + E)) \), where \( \alpha \) is the inverse of Ackermann’s function, a very slow-growing function that is typically described as “almost linear.”

### 3.4 Interval Graphs

Interval graphs are a class of graphs that have efficient polynomial-time solutions for many problems that are NP-Complete for general graphs. Let \( G = (V, E) \) be an undirected graph. There are at least four equivalent definitions of interval graphs [13, 16, 19]; here, only two are needed.

**Definition 1.** \( G \) is an interval graph if it is the intersection graph of some set of intervals on the real number line. [19]

For Definition 2, a clique in a graph is a complete subgraph, i.e. a subset \( V' \subseteq V \) such that there is an edge between pair of vertices in \( V' \). \( V' \) is a maximal clique if there is no subset of vertices \( V'' \supseteq V' \) such that \( V'' \) is also a clique.

**Definition 2.** \( G \) is an interval graph if all of its maximal cliques can be ordered such that for each vertex \( v \), all of the cliques containing \( v \) occur consecutively in the ordering [13].

Fig. 3 illustrates Definitions 1 and 2 of interval graphs. Fig. 3 (a) shows a set of overlapping intervals and Fig. 3 (b) shows their intersection graph. The vertical lines in Fig. 3 (a) represent the maximal cliques. The left-to-right ordering of maximal cliques satisfies Definition 2.

![Figure 3. A set of intervals with vertical lines representing maximal cliques (a) and corresponding interval graph (b).](image)

### 4. SSI Properties

Here, we derive several properties of SSI Form that will be useful throughout the remainder of the paper.

**Lemma 1.** For a procedure in SSI form, the definition of each variable dominates its uses, and each use post-dominates the definition.

**Proof.** Property 5.3 of Ananian’s MS Thesis [2, p. 20] states that every definition of a variable dominates all of its non-\( \phi \)-function uses and every use post-dominates any non-\( \sigma \)-function definitions. We extend this property to include \( \phi \)-function uses and \( \sigma \)-function definitions.

Consider a generic \( \phi \)-function \( y \leftarrow \phi(..., x, ...). \) Let \( n \) be the basic block containing the \( \phi \)-function, and suppose that \( m \) is the predecessor of \( n \) from which the value of \( x \) flows into \( y \). The use of \( x \) at the \( \phi \)-function is associated with \( m \), not \( n \). Clearly, \( x \) is live inside of \( m \), since it is certainly possible that \( x \) is used in \( m \). Therefore, the definition point of \( x \) dominates \( m \), and consequently, the use of \( x \) associated with the \( \phi \)-function. This ensures that every definition of a variable dominates it uses, regardless of whether the use is by an instruction or a \( \phi \)-function. The same basic argument holds for \( \sigma \)-function definitions, since the actual definitions are associated with the successors of the basic blocks containing the \( \sigma \)-function.

**Lemma 2.** Consider a procedure in SSI form. Consider a variable \( v \) that is defined at point \( p \) and used at points \( m \) and \( n \) in the program. Then either \( n \ spdom m \) or \( m \ spdom n \).

**Proof.** Recall that \( t \) is the exit node of the CFG. First, assume to the contrary that neither \( n \ spdom m \) nor \( m \ spdom n \). Then there is no path \( p \rightarrow n \rightarrow t \) that excludes \( m \) and \( p \rightarrow m \rightarrow t \) that excludes \( n \). Therefore neither \( n \ spdom p \) nor \( m \ spdom p \), which contradicts Lemma 1. The procedure is not in SSI form.

A death point is defined to be a use of a variable in an instruction where the variable is no longer live following the instruction. Given \( LIVEOUT(n) \), the death points occurring in block \( n \) can easily be marked by traversing the basic block in reverse order [8, p. 640].

**Corollary 1.** Each variable \( v \) in a procedure in SSI form has exactly one death point.

**Proof.** Follows immediately from Lemma 2. The death point of variable \( v \) is simply the unique use that post-dominates all other uses of \( v \), as well as \( v \)-s definition.
**Corollary 2.** For a procedure in SSI form, variable \( v \) is live at point \( p \) in the program if and only if the definition of \( v \) dominates \( p \) and the death point of \( v \) post-dominates \( p \).

**Proof.** Follows immediately from Lemma 2 and Corollary 1. \( \Box \)

## 5. POST-DOMINATED DFS

A Post-Dominated Depth First Search (PD-DFS) of a CFG is a Depth-First Search (DFS) where each node \( n \) is processed only after every node \( m \) such that \( n \) post-dominates \( m \) is processed. If the PD-DFS is initiated from \( r \), the CFG entry node, and each basic block is traversed in forward order, then the definition point of variable \( v \) will be processed before any other statement where \( v \) is live by Corollary 2. Likewise, Corollary 2 ensures that the PD-DFS processes the death point after all other points during which the variable is live.

Fig. 4 provides a brief example that illustrates a PD-DFS. Fig. 4 (a) shows a CFG fragment, Fig. 4 (b) shows the post-dominator tree, and Fig. 5 (c) shows legal results of a PD-DFS. As a counterexample, a sequence of nodes starting with \( A, C, D, \ldots \), would not be a PD-DFS, because \( D = \text{idom} B \), and \( D \) would occur before \( B \) in this sequence.

Fig. 5 shows pseudocode that computes a PD-DFS. Like the traditional implementation of a DFS, the PD-DFS uses a stack. Unlike a traditional DFS, the PD-DFS may stop and backtrack rather than processing a new successor \( s \) of the current node \( n \). If some node \( m \) such that \( s = \text{idom} m \) has not yet been processed, then the PD-DFS will not immediately push \( s \) onto the stack; instead, the search will backtrack, and process \( s \) at a later point.

As an example, consider an if-then-else statement. Let \( a \) be the block containing the condition, \( b \) and \( c \) be blocks representing the two sides of the condition, and \( d \) be the block that merges the two paths of control. Note that \( d = \text{idom} a = \text{idom} b = \text{idom} c \). First \( a \) is processed and pushed onto the stack. Without loss of generality, suppose that \( b \) is processed and pushed next. A traditional DFS would process \( d \) next, since \( d \) is a successor of \( b \); however, \( c \) has not yet been processed. This causes the PD-DFS to backtrack. Since \( b \) has no other successor, \( b \) is popped from the stack. With \( a \) on top of the stack, \( c \) is processed next. Since \( d \) is a successor of \( c \) as well, \( c \) can be processed after \( c \), since all nodes post-dominated by \( d \) have now been processed.

**Figure 4.** A CFG fragment (a), post-dominator tree (b), and the two legal results of the PD-DFS (c).

**Figure 5.** Pseudocode to build a list of basic blocks in PD-DFS order.

In Fig. 5, variable \( \text{idom} \_c[...] \) is an array of integers. \( \text{idom} \_c[n] \), which is initialized to 0, is the number of nodes \( m \), such that \( n = \text{idom} m \), that have already been processed. When \( \text{idom} \_c[n] \) is equal to the number of children of \( n \) in \( P \), the post-dominator tree, \( n \) can finally be processed. It goes without saying that this criterion also implies that every node that is post-dominated by \( n \) has also been processed.
A node, selected in line 33, is Lemma 3. Lemma will be applied in Section 7.

Let FORM ARE INTERVAL GRAPHS.

There is no guarantee that a PD-DFS initiated from the root will uncovered every node without stopping at least once. This is why there is an outer do-while loop in Fig. 5 spanning lines 8-37. As an example, consider the CFG shown in Fig. 6 (a). There are 6 different PD-DFS solutions, as shown in Fig. 6 (n). In solutions \[A, C, D, B, E\] and \[A, B, D, E\], the inner-loop terminates after processing block \(B\). Since \(E\) is not reachable from \(B\), the condition in line 32 (and then again in line 36) is satisfied, and the outernost loop executes again. In each case, the only unmarked node, selected in line 33, is \(E\).

A Contiguous PD-DFS (CPD-DFS) is similar in principle to a PD-DFS. The primary difference, however, is that one extra condition is placed on basic blocks before they can be processed. In a PD-DFS, block \(n\) can only be processed after every block \(m\) such that \(n\) spdom \(m\) is processed. In addition to this condition, a CPD-DFS requires that every predecessor of block \(n\) (other than \(n\) itself) be processed prior to \(n\). Referring back to Fig. 6 (b), only \[A, B, C, D, E\] and \[A, B, D, C, E\] can be classified as a CPD-DFS, since \(B\) is a predecessor of both \(C\) and \(D\).

This issue will become more prevalent in Section 7 when we discuss liveness analysis. It is not relevant to the proof that interference graphs for SSI form procedures are interval graphs; for this proof, a PD-DFS suffices.

Let \(G = (N, E, r, t)\) be a CFG and \(L\) be a list of CFG nodes in CPD-DFS order. Let \(L(n) = i\) if node \(n\) is the \(i^{th}\) node in the list. Finally, let \(m, n \in N\) and \(S = (m = n_1, ..., n_k = n)\) be a path with no repeating vertices (i.e. loops) from \(m\) to \(n\) in \(G\). The following Lemma will be applied in Section 7.

Lemma 3. For \(n_1, 1 \leq i \leq k-1, L(n_i) < L(n_{i+1})\).

Proof. The proof is trivial. Let \(S_{i,j} = (m, n_2, ..., n_j)\) and use induction on \(j\). Since \(n_{i+1}\) is a predecessor of \(n_j\), then \(L(n_{i+1}) < L(n_j)\) for all \(j\).

6. INTERFERENCE GRAPHS IN SSI FORM ARE INTERVAL GRAPHS

Here, we prove that the interference graph for a procedure in SSI Form is an interval graph using a PD-DFS. Let \(Interval[x].start\) and \(Interval[x].end\) respectively represent the birth and death points of variable \(x\). To construct the interval for each variable, initialize a counter, \(index\), to 0. \(Interval[p].start = 0\) for all parameters \(p\).

As each basic block \(n\) is processed, increment \(index\). Next, for every variable \(y\) that is defined by an \(\sigma\)-function \((..., y, ...) \leftarrow \sigma(x)\) that flows into \(n\), \(Interval[y].start\) is set to \(index\). Next, \(index\) is incremented and each instruction is processed. First, each variable \(x\) that is used by the instruction is processed. If the current instruction is a death point of \(x\), then \(Interval[x].end\) is set to \(index\). Index is incremented again, in order to ensure that the intervals associated with variables defined by the current instruction do not interfere with variables that die there. For each variable \(y\) defined by the current instruction, \(Interval[y].start\) is set to \(index\). After the last instruction is processed, \(index\) is incremented again. Each variable \(x\) that is used by a \(\varphi\)-function \(y \leftarrow \varphi(..., x, ...)\) that flows out of \(n\) dies immediately; so \(Interval[x].end\) is set to \(index\).

Theorem 1. The interference graph for the procedure in SSI Form is an interval graph.

Proof. To see that the above algorithm is correct, assume to the contrary that for some variable \(x\), there is some value \(i\), \(Interval[x].start < i < Interval[x].end\), where \(x\) is not live at the point \(p\) in the program corresponding to \(i\). By the construction of PD-DFS, the definition point of \(x\) dominates \(p\), and the death point of \(x\) post-dominates \(p\). By Corollary 2, \(x\) is live at \(p\)—a contradiction. Since the live range for each variable corresponds to an interval. The interference graph is therefore the intersection graph of a set of intervals, and is therefore an interval graph by Definition 1.

Figs. 7 and 8 show pseudocode for a procedure that builds intervals for each variable, based on Theorem 1. It is important to note that this pseudocode performs liveness analysis and marks death points up front. The next section addresses liveness analysis in more detail.

The pseudocode in Figs. 7 and 8 also outputs the maximal cliques in accordance with Definition 2 of an interval graph. To ensure that the cliques that are output are maximal, the variables that are currently live (LIVENOW) are output immediately before a death point—upon which a variable is removed.

A clique is only output at a death point if at least one variable has been defined by an instruction between the previous death point and the current one (the Boolean variable \(max\ clique\) ensures this). Admittedly, using Figs. 7 and 8 to verify both Definitions 1 and 2 of an interval graph is somewhat redundant; however, we have included both to ensure completeness of the presentation.

The pseudocode in Figs. 7 and 8 could easily be modified to output an interference graph rather than the maximal cliques. When each newly-defined variable is encountered, a vertex is allocated to that variable. Edges are added between the currently defined variable and all variables in the LIVENOW set.

It is clear from Theorem 1 that the live range of each variable \(v\) corresponds directly to \([Interval[v].start, Interval[v].end]\). The interference graph could alternatively be constructed by computing the intersection of each pair of intervals in accordance with Definition 1 of interval graphs.
We can ascertain that the use is not a death point if we encounter another use of $v$—by Corollary 1, $v$ has exactly one death point. Alternatively, we can ascertain that the use is a death point if we eventually encounter $i$, the exit node of the CFG, without encountering another use of $v$.

To eliminate this conundrum, it suffices to traverse the CFG in Reverse PD-DFS (RPD-DFS) order. The first use of each variable encountered will be the death point and the variable becomes live. Likewise, the variable is no longer live once we encounter its definition point, which is immediately clear. Since each variable is defined once in SSI Form, this conundrum is unnecessary. If the CFG is traversed in RPD-DFS order, it also necessary to process each basic block $n$ in reverse order: $\sigma$-functions flowing out of $n$ are processed first, then the instructions within $n$ are processed in reverse order, and only the $\varphi$-functions in $n$.

In the remainder of this section, we show that liveness analysis requires one iteration to converge to the correct solution for a CFG in SSI Form. Specifically, the CFG must be traversed in Reverse CPD-DFS (RCPD-DFS) order. The construction of live intervals and/or an interference graph can proceed at the same time, thereby eliminating the need to call lines 1 and 2 in Fig. 7.

7. LIVENESS ANALYSIS IN SSI FORM

Lines 1 and 2 of Fig. 7 perform liveness analysis and compute the death points of each variable. From Theorem 1, the live range for variable $v$ is precisely $[\text{interval}[v].\text{start}, \text{interval}[v].\text{end})$. By traversing the CFG in PD-DFS order, we ensure that $\text{interval}[v].\text{start}$ is the first point within the interval that is processed, and $\text{interval}[v].\text{end}$ is the last.

Now, suppose that we come to a use of variable $v$ during the PD-DFS traversal of the CFG. Without running liveness analysis in advance, we cannot determine whether or not this use is a death-point of $v$. We can ascertain that the use is not a death point if we encounter another use of $v$—by Corollary 1, $v$ has exactly one death point. Alternatively, we can ascertain that the use is a death point if we eventually encounter $i$, the exit node of the CFG, without encountering another use of $v$.

To eliminate this conundrum, it suffices to traverse the CFG in Reverse PD-DFS (RPD-DFS) order. The first use of each variable encountered will be the death point and the variable becomes live. Likewise, the variable is no longer live once we encounter its definition point, which is immediately clear. Since each variable is defined once in SSI Form, this conundrum is unnecessary. If the CFG is traversed in RPD-DFS order, it also necessary to process each basic block $n$ in reverse order: $\sigma$-functions flowing out of $n$ are processed first, then the instructions within $n$ are processed in reverse order, and only the $\varphi$-functions in $n$.
7.1 A New Liveness Equation for SSI Form

For the purposes of theoretical correctness, we will prove that an iterative dataflow solver converges in exactly one iteration for SSI Form. Practically speaking, a second iteration of a standard iterative solver would be required to ensure that none of the \text{LIVEOUT} sets change; however, the proof ensures that the second iteration is unnecessary.

Before doing so, however, it is necessary to make a small modification to Eq. (1) for liveness analysis in order to properly account for \varphi- and \sigma-functions. The modification for \varphi-functions have already been described previously [5].

First and foremost, any variable defined by a \varphi- or \sigma-function in basic block \( n \) is added to \text{VARKILL}(n). Since the definitions occur in \( n \), these variables should not be propagated to \( n \)'s predecessors.

Secondly, the set \text{UEVAR}(m) is replaced. \text{UEVAR}(m) is the set of variables that are “upwards exposed” from \( m \) to all of its predecessors. Using a non-SSA/non-SSI CFG, \( m \) would propagate the same set of variables to all of its predecessors. \varphi- and \sigma-functions, however, cause \( m \) to propagate different variables to its different predecessors. Therefore, \text{UEVAR}(m) is replaced with a different set, \text{UEVAR}(m, n), the set of upwards-exposed variables from \( m \) to \( n \), where \( n \) is a predecessor of \( m \).

For example, let \( n_1 \) and \( n_2 \) be predecessors of block \( m \), and suppose that \( m \) contains a \varphi-function \( y_1 = \varphi(x_1, x_2) \) where \( x_1 \) and \( x_2 \) flow from \( n_1 \) and \( n_2 \) respectively. Then \( x_1 \) is placed into \text{UEVAR}(m, n_1) and \( x_2 \) is placed into \text{UEVAR}(m, n_2). As discussed previously, \( y_1 \) is then placed into \text{VARKILL}(m).

Now, consider a basic block \( n \) with successors \( m_1 \) and \( m_2 \). Suppose that \( n \) contains a \sigma-function \( y_1 \leftarrow \sigma(x_1) \), where \( x_1 \) flows from \( n \) to \( m_1 \) and \( y_2 \) flows from \( n \) to \( m_2 \). Then \( x_1 \) is placed into \text{UEVAR}(m_1, n) and \text{UEVAR}(m_2, n); meanwhile, \( y_1 \) is placed into \text{VARKILL}(m_1) and \( y_2 \) is placed into \text{VARKILL}(m_2).

Finally, suppose that variable \( v \) is used by an instruction in block \( m \), but is not defined (either by an instruction or a \varphi-function). Then \( v \) is placed in \text{UEVAR}(m, n_1) for each predecessor \( n_1 \) of \( m \).

Given these modifications, \text{UEVAR}(m) in Eq. (1) is replaced with \text{UEVAR}(m, n).

7.2 Liveness Analysis for Procedures in SSI Form Requires One Iteration To Converge

Here, we formally prove that liveness analysis requires a single iteration to converge. For the purposes of proving that the interference graph for SSI Form programs is an interval graph, a traversal of the nodes in PD-DFS or RPD-DFS order suffices; however, for liveness analysis, RCPD-DFS order is required.

To illustrate the necessity of RCPD-DFS, refer to the CFG in Fig. 6 (a). Suppose that some variable \( v \) is defined in basic block \( A \) and is used once, in block \( E \). \( v \) therefore belongs to \text{LIVEOUT} sets of blocks \( A, B, C, \) and \( D \). Due to the use, \( v \) will also be placed in the \text{UEVAR} sets for block \( E \): \text{UEVAR}(E, C) and \text{UEVAR}(E, D).

This will propagate \( v \) from \( E \) to \( C \) and \( D \) during the first iteration of liveness analysis.

Now, if the nodes are processed in RPD-DFS order, one possible ordering according to Fig. 6 (b) is \([E, B, C, D, A]\). Initially, all of the \text{LIVEOUT} sets are empty. First, \( E \) is processed; since the death point of \( v \) occurs in \( E, v \) is not added to \text{LIVEOUT}(E). Next, \( B \) is processed. Based on Eq. (1) (as modified in Section 7.1), the variables in \text{LIVEOUT}(C) and \text{LIVEOUT}(D) that do not belong to \text{VARKILL}(C) and \text{VARKILL}(D) respectively are propagated into \text{LIVEOUT}(B). Since both \text{LIVEOUT}(C) and \text{LIVEOUT}(D) are empty, \text{LIVEOUT}(B) remains empty. Processing \( C \) and \( D \) sequentially propagates \( v \) from \text{LIVEOUT}(E) to \text{LIVEOUT}(C) and \text{LIVEOUT}(D) respectively, finally \( v \) is propagated to \text{LIVEOUT}(A). A second iteration of liveness analysis is required to propagate \( v \) to \text{LIVEOUT}(B) from \text{LIVEOUT}(C) and/or \text{LIVEOUT}(D). If RCPD-DFS is used, one the other hand, block \( C \) and/or block \( D \) will be processed prior to \( B \), therefore \( v \) will be propagated to \text{LIVEOUT}(B) during the first iteration.

The following Theorem formally proves that a single iteration is required for liveness analysis. We assume that the SSI Form CFG has been constructed in advance, and that the CFG has been traversed once in order to compute the \text{UEVAR} and \text{VARKILL} sets for each basic block.

**Theorem 2.** For a program in SSI Form, liveness analysis terminates after one iteration through the nodes of a CFG if basic blocks are processed in RCPD-DFS order.

**Proof.** Suppose that variable \( v \) is placed in the set \text{LIVEOUT}(m) by an iterative dataflow analyzer that performs liveness analysis. Then there are three possibilities:

1. There is a path from the definition of \( v \) to the end of block \( n \) and a path from the end of \( n \) to use of \( v \).
2. \( v \) is a parameter of a \varphi-function (flowing from \( n \)) into at least one successor of \( n \).
3. \( v \) is the parameter of a \sigma-function in \( n \).

Case (1) is justified by Lemma 3. Let \( n_1 \) be the basic block containing the use of \( v \). Then there is a path \( S \) in the CFG from \( n \) to \( n_1 \) containing no repeating CFG nodes such that the vertices in \( S \) are taken in reverse order by the RCPD-DFS. Specifically, if \( S = (n = n_1, …, n_{k-1}, n_1 = n_1) \), first \( n_1 \) is processed; next \( n_{k-1} \) is processed, and \( v \) is propagated from \( n_1 \) to \( n_{k-1} \); and the process continues, inductively, until \( v \) is propagated from \( n_2 \) to \( n_1 \). Therefore \( v \) is placed into \text{LIVEOUT}(n_1) during the first iteration.

With respect to case (2), let \( m \) be the successor of \( n \) contains a \varphi-function \( \ldots \leftarrow \varphi(\ldots, v, \ldots) \), where \( v \) flows from \( m \) to \( n \). Then \( v \) is placed into \text{UEVAR}(m, n) during the initial traversal of the CFG and is propagated from \( m \) to \( n \) during the first iteration.

With respect to case (3), let \( m_1, \ldots, m_l \) be successors of \( n \), and suppose that \( n \) includes a \sigma-function \( \ldots \leftarrow \sigma(v) \) that defines a new variable in each \( m_i \). Then for \( 1 \leq i \leq j \), \( v \) is placed into \text{UEVAR}(m_i, n). \( v \) is therefore propagated from each \( m_i \) to \( n \) during the first iteration. □

7.3 Discussion

It is interesting to note that Theorem 2 would actually hold if the blocks were processed in Breadth-First Search (BFS) order, starting at the root of the CFG. The reason that the RCPD-DFS order is used is so the live intervals and/or an interference graph can be constructed at the same time.

Also, the liveness analysis equation (Eq. (1), as modified in Section 7.1) does not need to be computed exactly. Although the \text{LIVEOUT} sets are required to be computed, there is no need to
explicitly compute the VARKILL and UEVAR sets. When processing block \( m \) in reverse order, the LIVENOW set will be the LIVEMIN set after processing the block in reverse order. Processing the instructions in the block in reverse order will have effectively filtered all of the variables belonging to the VARKILL set. By examining the \( \sigma \)-functions in \( m \) and the \( \sigma \)-functions in each predecessor \( n \) of \( m \), the correct variables can be propagated backwards into LIVENOW without explicitly constructing UEVAR(\( m, n \)).

Before processing a CFG node \( n \), the \( \sigma \)-functions in each successor \( m \) should also be processed, to correctly handle loops. If \( m \) is the header of a loop, then \( m \) may be the last node in the loop. Any variable used as a \( \sigma \)-function parameter in \( m \) that flows from \( n \) should be added to LIVENOW(\( n \)) implicitly. \( \sigma \)-functions of the form \( v \leftarrow \sigma (\ldots, v, \ldots) \) often occur in loops. Examining these \( \sigma \)-functions is the only way to propagate \( v \) into the bottom of the loop if UEVAR is not explicitly constructed.

8. CONCLUSION

We have shown that an interference graph for a procedure represented in SSI Form is an interval graph. We have also shown that liveness analysis for an SSI-form procedure can provably converge in one iteration of a standard dataflow solver.

We envision two potential areas of future research in this area. The first is to develop a linear scan implementation of register allocation for SSI-form programs. The advantage over this technique compared to prior implementations of linear scan is that there will be no lifetime holes. The potential drawbacks include the cost of constructing and deconstructing SSI Form. In principle, the translation out of SSI Form following register assignment should be no different than that of SSA Form—both \( \sigma \)- and \( \sigma \)-functions represent parallel copies in slightly different contexts. Consequently, techniques such as those proposed by Hack et al. [17-18] for \( \sigma \)-functions can be adopted directly for \( \sigma \)-functions as well.

The second area of future research is to develop a register allocation framework for SSI that solves the \( k \)-colorable subgraph problem directly on an interval graph. To date, we are unaware of any register allocation techniques that solve this problem directly.

REFERENCES


