Interference Graphs for Procedures in Static Single Information Form are Interval Graphs

Philip Brisk
Ecole Polytechnique Federale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
philip.brisk@epfl.ch

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

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 SSI 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. G.2.2 [Graph Theory]: Graph Algorithms.

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, 24, 27, 29, 31] 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, 25].

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 primary consequences of this result are:

1. It is possible to use SSI Form to implement a linear scan register allocator that does not have any lifetime holes.
2. The $k$-colorable subgraph problem can be solved in polynomial time for interval graphs; this can be used to implement spilling in a graph coloring register allocator.
3. In SSI Form, liveness analysis can converge in one iteration if the basic blocks of an application are processed in a certain order.

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 [4-5, 17]. 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].

The theoretical significance of this result, as pointed out by Brisk et al. [5] is that the minimum number of registers required to avoid spilling all scalar variables can be computed efficiently. The primary applications for this result are high-level synthesis and embedded processor design, where automated tools can be used to generate a customized architecture. Of course, the number of registers required to meet this constraint may change if the program is transformed using optimizations such as loop unrolling or software pipelining that effectively rewrite the application.

None of the aforementioned theoretical problems, however, corresponds directly to any of the problems faced by register allocation in software compilation: 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.
On the other hand, 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 instructions required to translate a procedure out of SSA form [18, 25]. Pereira and Palsberg [23], have further proven that register allocation is NP-Complete following the translation out of SSA Form. Hack et al. [17, 18], on the other hand, have shown how to eliminate SSA Form following register assignment while preserving the assignment of variables to registers.

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 reasonable, 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 $V^c = V'c$ 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. In the graph-theoretic version of $k$-colorable subgraph, in contrast, those vertices not in the subgraph are removed completely from the resulting graph. This dichotomy is one challenge that must be faced when implementing a register allocator that performs spilling by solving the $k$-colorable subgraph problem.

For chordal graphs, the $k$-colorable subgraph problem is NP-Complete if $k$ is unbounded, and polynomial in $k$ if $k$ is a constant [32]. In other words, this means that the runtime of the problem will be of the order $O(n^k)$; for a target architecture with 32 registers, for example, this means that the running time will be $O(n^{32})$. In this case, polynomial-time 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 [32], 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]. We believe that it is possible to design and implement a register allocator that solves this problem directly for interval graphs.

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, 29, 31].

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 drawback—lifetime holes.

2.2 Linear Scan Register Allocation

A second advantage of using SSI Form involves the linear scan family of register allocation algorithms [21, 24, 27, 29, 31]. 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, 29, 31].

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 drawback—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.

For each block $b$ in a CFG, $\text{LIVEOUT}(b)$ is defined to be the set of variables that are live at the exit point of $b$. Given $\text{LIVEOUT}(b)$, the set of variables that are live at each point within $b$ can easily be determined by traversing the instructions within $b$ 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 [8]. Before describing this process we introduce two additional sets for each basic block:

$\text{UEVAR}(n)$ 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 in $n$. Therefore, they will belong to the $\text{LIVEOUT}$ sets of each predecessor of $n$.

$\text{VARKILL}(n)$ 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 $\text{LIVEOUT}(n)$ is initially empty. Then, the following equation updates $\text{LIVEOUT}(n)$.

\[
\text{LIVEOUT}(n) = \text{LIVEOUT}(n) \cup \bigcup_{m: \text{succ}(n)} \left(\text{UEVAR}(m) \cup \text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}\right)
\]

This equation is solved for each basic block $n$—this process is called an iteration. If the $\text{LIVEOUT}$ set changes for at least one block, then another iteration is performed. The process repeats until none of the $\text{LIVEOUT}$ sets change 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 \( O(I + VN^2) \); 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, 26, 30] 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.

![Figure 1. A CFG fragment (a) in SSA form (b)](image)

The semantics of the \( \phi \)-function are as follows: if the left path is taken into the block that uses \( v \), then \( v_1 \) receives the value of \( v_3 \); if the right path is taken, then \( v_2 \) receives the value of \( v_2 \).

\( \phi \)-functions are placed at confluence points, where multiple paths in a CFG merge. A \( \phi \)-function 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 \( v_1 \); likewise, the use of \( v_2 \) is associated with the end of the block that defines \( v_2 \).

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 [28]. SSI Form is an extension of SSA Form. SSI Form uses \( \sigma \)-functions in the same way as SSA Form. In addition to \( \phi \)-functions, SSI form inserts \( \sigma \)-functions at split points, such as conditional branches. A \( \sigma \)-function merges multiple variables into one unique variable name; a \( \sigma \)-function, in contrast, splits one variable into multiple ones at a branch point—alogous 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. In Fig. 2 (c), a \( \sigma \)-function is inserted at the end of 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 $n \text{ dom } m$) if every path from $r$ to $m$ passes through $n$. By convention, every node dominates itself, i.e., $n \text{ dom } n$ is always true. $n$ strictly dominates $m$ (denoted $n \text{ sdom } m$) if $n \text{ dom } m$ and $n \neq m$. $n$ post-dominates $m$ (denoted $n \text{ pdom } m$) if every path from $m$ to $t$ passes through $n$. $n$ strictly post-dominates $m$ (denoted $n \text{ spdom } m$) if $n \text{ sdom } m$ and $n \neq m$.

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

Every node in the CFG has an immediate dominator, except for $r$, 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 $E_D = \{(n, \text{idom } n) \mid n \in N - \{r\}\}$ 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 $E_P = \{(n, \text{ipdom } n) \mid n \in N - \{t\}\}$ 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 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].

**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 (maximal) cliques containing $v$ occur consecutively in the ordering [13].

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.

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.

### 4. SSI Properties

Here, we derive several properties of SSI Form that will be useful throughout the remainder of the paper. Let $D_v$ be the definition of variable $v$ and $U_v$ be a use of $v$.

**Lemma 1.** For an SSI Form procedure, $D_v, \text{ dom } U_v$ and $U_v, \text{ pdom } D_v$ for every use $U_v$ of variable $v$.

**Proof.** Property 5.3 of Ananian’s MS Thesis [2, p. 20] states that every $D_v$ dominates every use $U_v$, that is not a $\varphi$-function use and every use $U_v$ post-dominates any definition $D_v$, that is not a $\sigma$-function definition of $v$. This property is extended to include $\varphi$-function uses and $\sigma$-function definitions.

Consider a generic $\varphi$-function $y \leftarrow \varphi(x, \ldots, \ldots)$. Let $n$ be the block containing the $\varphi$-function, let $m$ be the predecessor of $n$ from which $x$ flows into $y$. Use $U_v$ of $x$ in the $\varphi$-function is associated with $m$, not $n$. Clearly, $x$ is live in $m$. Thus $D_v, \text{ dom } U_v$ [17], regardless of whether $U_v$ is a use of $x$ as a $\varphi$-function parameter.

The same argument holds for $\sigma$-function definitions. The actual definitions are associated with the successors of the basic blocks containing the $\sigma$-function; the proof uses analogous reasoning. □
Lemma 2. Consider a procedure in SSI form. Let $D_v$ be the definition of $v$ and $U_1$ and $U_2$ be uses of $v$. Then $U_1 \text{ pdom } U_2$ or $U_2 \text{ spdom } U_1$.

**Proof.** Recall that $t$ is the exit node of the CFG. First, assume to the contrary that neither $U_1 \text{ spdom } U_2$ nor $U_2 \text{ spdom } U_1$. Then there are paths $D_v \rightarrow U_1 \rightarrow t$ that excludes $U_2$ and $p \rightarrow U_2 \rightarrow t$ that excludes $U_1$. Hence, neither $U_1 \text{ spdom } D_v$ nor $U_2 \text{ spdom } D_v$, contradicting Lemma 1. The procedure cannot be in SSI Form.

A death point is a use of a variable $v$ such that $v$ is no longer live following the instruction. Given $\text{LIVEOUT}(n)$, 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.** By Lemma 2, the death point of $v$ is the unique use that post-dominates all other uses of $v$, as well as $v$'s definition.

**Corollary 2.** For an SSI Form procedure, $v$ is live at point $p$ if and only if $D_v \text{ dom } P$ and for death point $U_v \text{ pdom } p$.

**Proof.** Follows immediately from Lemma 2 and Corollary 1.

5. POST-DOMINATED DFS

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

Fig. 4 shows a CFG (a), post-dominator tree (b), and 2 different PD-DFSes (c) for the CFG. A sequence of nodes $A, C, D, ..., B$, would not be a PD-DFS, because $D = \text{ ipdom } B$ and would occur before $B$.

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 $n$.

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

![Figure 5. Pseudocode to build a list of basic blocks in PD-DFS order.](image-url)

Procedure: PD-DFS(G, P)

Input: CFG: $G = (N, E, r, t)$
Post-Dom. Tree: $P = (N, E_{P}, t)$

Local Vars:
- Stack: Stack
- Array of Integer: ipdom_c[1..|N|]
- Array of Boolean: Mark[1..|N|]
- CFG Node: $n$, $p$, $s$
- Integer: num_marked
- Boolean: all_succs_marked

Output: List of Vars: L

1. Let L be an empty list of variables
2. For each CFG node $n \in N$
   3. ipdom_c[n] ← 0
   4. Mark[n] ← False
   5. EndFor
   6. num_marked ← 0
   7. Stack.push(r)
   8. Do
      9. While !Stack.empty()
         10. $n \leftarrow$ Stack.top()
         11. If Mark[n] = False
            12. Mark[n] ← True
            13. num_marked ← num_marked + 1
            14. L.insert_back(n)
            15. If $n \neq t$
               16. $p \leftarrow$ P.get_parent(n)
               17. ipdom_c[p] ← ipdom_c[p] + 1
               18. EndIf
            19. EndIf
            20. all_succs_marked ← True
            21. For each successor $s$ of $n$
               22. If Mark[s] = False AND ipdom_c[s] = P.get_num_children(s)
                  23. Stack.push(s)
                  24. all_succs_marked ← False
               25. EndIf
               26. EndFor
            27. If all_succs_marked = True
               28. Stack.pop()
               29. EndIf
            30. EndWhile
               31. If num_marked < |N|
               32. Pick node $n$ with Mark[n] = False AND ipdom_c[n] = P.get_num_children(n)
               33. Stack.push (n)
               34. EndIf
            35. While num_marked < |N|
            36. Return L

If some node $m$ such that $s = \text{ ipdom } m$ 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. An example considers 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, d can be processed after c, since all nodes post-dominated by d have now been processed.

In Fig. 5, variable ipdom_c[...] is an array of integers. ipdom_c[n], which is initialized to 0, is the number of nodes m, such that n = ipdom m, that have already been processed. When ipdom_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.

6. INTERFERENCE GRAPHS IN SSI FORM ARE INTERVAL GRAPHS

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 σ-function (…, y, ...) ← σ(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. 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 σ-function y ← σ(..., x, ...) that flows out of n dies immediately; so Interval[x].end is set to index.

A variable x whose death point is the current instruction does not interfere with a variable y defined by the current instruction. The reason is that the intervals are closed on the left and open on the right. In other words, for x, the interval would be [Defx, index) and for y the interval would be [index, Deathy), where Defx and Deathy are the respective definition and death points of x and y.

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 provide pseudocode that builds intervals for each variable and outputs the maximal cliques in accordance with Definitions 1 and 2 of an interval graph. This is somewhat redundant before both definitions are equivalent; however, both have been included to ensure completeness of the presentation.

To ensure that the cliques that are output are maximal, the variables that are currently live (LIVENOW) are output just 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).

Procedure: Build_Intervals_and...

Output: Array of [Integer : start, Integer : end] : Interval[1..|V|]

Figure 7. Pseudocode that builds intervals and outputs maximal cliques for an SSI-form procedure (continued in Fig. 8)
To eliminate this issue, we can also traverse the CFG in Reverse PD-DFS (RPD-DFS) order. The first use of \( v \) that is encountered will be its death point and becomes live. Likewise, \( v \) is no longer live once we encounter its definition point, since \( v \) is defined once in SSI Form. If the CFG is traversed in RPD-DFS order, it also necessary to process each basic block in reverse 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.

Section 7.1 describes how to handle \( \varphi \)- and \( \sigma \)-functions during liveness analysis. Section 7.2 defines a new liveness equation for SSI Form, which exploits Corollary 2. Section 7.3 shows that if the new liveness equation is used and basic blocks are process in RPD-DFS order, then liveness analysis converges in one iteration.

### 7.1 Handling \( \varphi \)- and \( \sigma \)-Functions

Here, we 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.

First and foremost, any variable defined by a \( \varphi \)-function in basic block \( n \) is added to \( \text{VARKILL}(n) \).

Second, let \( n_1 \) and \( n_2 \) be predecessors of block \( m \), and suppose that \( m \) contains a \( \varphi \)-function \( y \leftarrow \varphi(x_1, x_2) \) where \( x_1 \) into \( m \) from \( n_1 \) and \( x_2 \) flow into \( m \) from \( n_2 \). Clearly, \( x_1 \) and \( x_2 \) do not interfere in \( m \), and \( x_1 \in \text{LIVEOUT}(n_1) \) and \( x_2 \in \text{LIVEOUT}(n_2) \). To represent this fact, we let \( \varphi\text{-PARAM}(n, m) \) be the set of \( \varphi \)-function parameters that flow into \( m \) from \( n \). Trivially, any variable belonging to \( \varphi\text{-PARAM}(n, m) \) should be placed in \( \text{LIVEOUT}(n) \). This can be done during the initial traversal of the CFG that is used to build the \( \text{UEVAR} \) and \( \text{VARKILL} \) sets.

Now, we turn our attention to \( \sigma \)-functions. Let \( m \) be a basic block containing a \( \sigma \)-function \( y \leftarrow \sigma(x) \). The use of \( x \) is treated just like a normal use of \( x \) in block \( m \). Therefore, if \( x \) is not defined in \( m \), then \( x \) is added to \( \text{UEVAR}(m) \). Let \( n_1 \) and \( n_2 \) be the successors of \( m \) corresponding to the definitions of \( y_1 \) and \( y_2 \). Then \( y_1 \) is added to \( \text{VARKILL}(n_1) \) and \( y_2 \) is added to \( \text{VARKILL}(n_2) \).

### 7.2 A New Liveness Equation for SSI Form

Eq. (1), which is the standard equation for liveness analysis, propagates liveness information from the successors of basic block \( n \) backward to \( n \). In SSI Form, we can propagate liveness information backward from \( \text{ipdom } n \) to \( n \), rather than the successors of \( n \). This approach is justified by Corollary 2.

To simplify notation, let \( s = \text{ipdom } n \). Eq. (2) shows the new liveness equation that will be used for SSI Form. Theorem 2, which follows, establishes the correctness of this equation.

\[
\text{LIVEOUT}(n) = \bigcup_{m \text{ succ}(n)} \varphi\text{-PARAM}(n, m) \\
\cup \text{UEVAR}(s) \cup \left( \text{LIVEOUT}(s) \cap \text{VARKILL}(s) \right)
\]
Theorem 2. If $\text{LIVEOUT}(s)$ is known, Eq. (2) correctly computes $\text{LIVEOUT}(n)$ for a procedure in SSI Form.

Proof. Assume to the contrary that Eq. (2) is incorrect. Then there are two possibilities:

1. \exists variable $v$ that is live at $p$, but $v \notin \text{LIVEOUT}(n)$, or
2. \exists variable $v$ that is not live at $p$, but $v \in \text{LIVEOUT}(n)$.

First, assume that case (1) occurs. If $m$ is a successor of $n$, $v \notin \phi$-$\text{PARAM}(n, m)$; otherwise, $v$ would be inserted to $\text{LIVEOUT}(n)$ during the initial pass over the CFG to build $\text{UEVAR}$ and $\text{VARKILL}$ sets.

Let $D_v$ be the definition point of variable $v$. By Corollary 2, $D_v \in \text{dom} p$; it follows that $v \notin \text{VARKILL}(s)$. Likewise, by Corollary 2, there must be a use $U_v$ of $v$ such that $U_v \in \text{spdom} p$. Since $p$ is the exit point of $n$, it follows that $U_v \in \text{spdom} p$. If $U_v$ occurs in block $s$, then $v \in \text{UEVAR}(s)$ and $v$ is propagated into $\text{LIVEOUT}(n)$ by Eq. (2). Thus, $U_v \in \text{spdom} s$, and thus $v \in \text{LIVEOUT}(s)$. Hence $v$ is propagated to $\text{LIVEOUT}(n)$ by the last term, since we have already shown that $v \notin \text{VARKILL}(s)$. This contradicts the assumption that case (1) has occurred. If $v$ is live at $p$, then $v \in \text{LIVEOUT}(n)$.

Now, let us assume that case (2) occurs. Let us assume that variable $v$ is not live at $p$, but has been added to $\text{LIVEOUT}(n)$. First of all, $v$ cannot belong to the set $v \notin \phi$-$\text{PARAM}(n, m)$, for some successor $m$ of $n$; otherwise, $v$ would be live at $p$.

Now, let us suppose that $v \in \text{UEVAR}(s)$. By definition, this ensures that there is a use $U_s$ of $v$ contained in $s$ and $U_s \in \text{spdom} p$. If $D_v$ is the definition point of $v$, $D_v \in \text{spdom} s$ and $D_v \in \text{dom} p$. Therefore $v$ is live at $p$ by Corollary 2, a contradiction.

The only remaining possible way to propagate $v$ into $\text{LIVEOUT}(n)$ is to have $v \in \text{LIVEOUT}(s)$ and $v \notin \text{VARKILL}(s)$, which together imply that $D_v \in \text{spdom} s$. By Corollary 2, $\neg D_v \in \text{dom} p$ since $v$ is not live at $p$. Thus, every path from $n$ to $s$ must go through $D_v$. If $s$ is the basic block containing $D_v$, it follows that $d \in \text{spdom} n$. Therefore $s \in \text{spdom} d \in \text{spdom} n$, which contradicts the fact that $s = \text{idom} n$.

This contradicts the assumption that case (2) has occurred. If $v$ is not live at $p$, then $v \notin \text{LIVEOUT}(n)$. \qed

7.3 Liveness Analysis Converges in 1 Iteration

Here, we prove that liveness analysis converges in 1 iteration for an application in SSI Form if Eq. (2) is used for liveness analysis. Given the result of Theorem 2, the proof itself is relatively simple.

Let $N$ be the set of nodes in the CFG. Let $L: N \rightarrow \{1, 2, \ldots, |N|\}$ be a one-to-one and onto function that assigns a unique integer to each CFG node based on the order node $n$ is uncovered in an PD-DFS. In other words, $L(n) = i$ if $n$ is the $i^{th}$ node uncovered in the PD-DFS. If $r$ and $t$ are the entry and exit nodes, it is easy to see that $L(r) = 1$ and $L(t) = |N|$ respectively.

Corollary 3. Let $n \in N - \{r, t\}$ be a node in the CFG. Then $L(\text{idom} n) < L(n) < L(\text{idom} n)$.

Proof. Follows immediately from the definition of $L$ and the definition of a PD-DFS. \qed

Theorem 3. If Eq. (2) is used to compute liveness analysis for a program in SSI Form, then liveness analysis converges in one iteration if basic blocks are processed in RPD-DFS order.

Proof. The single iteration does not count the initial pass over CFG to build $\text{UEVAR}$ and $\text{VARKILL}$ sets. During this traversal, all variables belonging to $\phi$-$\text{PARAM}(m, n)$ and $\text{UEVAR}(s)$ can be added to $\text{LIVEOUT}(n)$. Therefore, the only vertices that we must consider in this proof are those that are propagated by the final term in Eq. (2).

We prove that one iteration suffices using induction on $L(n)$. Since RPD-DFS order is used, we count down from $|N|$ to 1.

For the basis, let $L(n) = |N|$. In this case, $n = t$, the exit node of the CFG. $t$ has no successors and $\text{LIVEOUT}(t)$ is empty. Therefore $\text{LIVEOUT}(t)$ is computed correctly during the first iteration.

For the induction hypothesis, suppose that $\text{LIVEOUT}(m)$ is computed correctly during the first iteration for all basic blocks $m$ such that $L(m) > k$.

For the induction, let $n$ be a basic block where $L(n) = k$. Moreover, let $s = \text{idom} n$. By Corollary 3, $L(s) > L(n)$. By the induction hypothesis $\text{LIVEOUT}(s)$ has been computed correctly during the first iteration. $\text{LIVEOUT}(n)$ is then computed correctly by Theorem 2. \qed

8. CONCLUSION AND FUTURE WORK

We have shown that an interference graph for a procedure represented in SSI Form is an interval graph. The primary consequences of this result are twofold: (1) It appears that it is possible to perform linear scan register allocation on an application in SSI Form without lifetime holes, thereby eliminating the need to perform a secondary optimization pass, such as second-chance bin packing [29] or interval splitting [41]; and (2) The spilling phase of a graph coloring register allocator for an application in SSI form can be implemented by solving the $k$-colorable subgraph problem directly, for an architecture with $k$ registers.

We have also shown that liveness analysis for an SSI-form procedure can provably converge in one iteration if a new SSI-specific equation for liveness analysis is used and basic blocks are processed in RPD-DFS order.

We envision two distinct areas of future work. First and foremost, we intend to implement an SSI-based linear scan register allocator and compare its results to a traditional linear scan allocator (e.g. [24]) along with an enhanced version with second-chance binpacking [29] and/or interval splitting [41]. Both the runtime and quality of solution of the allocator will be important. For an SSI-based allocator, we must account for the cost of translating the application out of SSI Form following register allocation, an issue that does not affect non-SSI-based allocators.

Second, we intend to implement an SSI-based graph coloring register allocator that solves the $k$-colorable subgraph problem directly for spilling, and compare it to a traditional graph coloring register allocator. To date, most allocators use a heuristic to solve the problem. Specifically, vertices are removed one-by-one from the interference graph and after each vertex is removed, the interference graph is tested for $k$-colorability using a heuristic. Neither the choice of vertex to be removed nor the heuristic used to test for $k$-colorability is optimal in the general case. Solving the $k$-colorable subgraph problem directly on an interval graph would
replace these heuristics with a polynomial-time algorithm that is optimal, in the graph theoretic sense.

Spilling, however, remains NP-Complete, even for straight-line code, based on a theoretical proof by Farach and Liberatore [10]. Their formulation, however, involves the linear-scan algorithm applied to a segment of straight-line code with no branches or loop structures. Their approach implicitly splits all live ranges. If a variable \( v \) is spilled, it is re-loaded into a register immediately before its next use. After that use, it remains in that register unless it is spilled again. The cost of the first spill of a variable includes the cost of the load instruction at the next use, plus the cost of the store instruction at the definition point of the variable; future spills of the same variable at points later will implicitly have a store cost of zero, since the store already exists. If all store costs are assumed to be zero implicitly, then the furthest-first approach advocated by Belady [3] is optimal; but when both load and store costs are taken into account, the problem becomes NP-Complete [10]: should the allocator spill a variable whose next use is furthest away, but has not been stored yet (positive store cost); or should it spill a variable whose next use is closer but has already been stored to memory (zero store cost)? The optimal answer will involve some type of exponential algorithm unless \( P = NP \).

Solving the \( k \)-colorable subgraph problem sidesteps this issue. When a variable is spilled, it is removed from the application, except immediately after the instruction that defines it and the instructions that use it. This formulation would not account for live-range splitting, which is one notable drawback. Therefore, a claim of an optimal solution to a graph theoretic optimization problem (\( k \)-colorable subgraph on an interval graph) cannot guarantee an optimal spilling/splitting solution. That being said, it is still potentially useful to investigate this new approach to SSI-based register allocation, and we intend to do so in the future.

ACKNOWLEDGEMENT

This work was influenced by discussions with Jens Palsberg, Fernando M. Q. Pereira, Sebastian Hack, Fabrice Rastello, and Florent Bouchez. The anonymous reviewers also provided many insights which improved the quality of the paper.

REFERENCES


