Optimal Register Sharing for High-Level Synthesis of SSA-Form Programs

Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh, Fellow, IEEE

Abstract—Register sharing for high-level synthesis of programs represented in Static Single Assignment (SSA) form is proven to have a polynomial-time solution. Register sharing is modeled as a graph coloring problem. Although graph coloring is NP-Complete in the general case, an interference graph constructed for a program in SSA form provably belongs to the class of chordal graphs, which have an optimal \(O(|V| + |E|)\)-time algorithm. Chordal graph coloring reduces the number of registers allocated to the design by as much as 86% and 64.93% on average compared to linear-scan register allocation.

Index Terms—High-level synthesis, Compilers (silicon)

I. INTRODUCTION

Register sharing during high-level synthesis is proven to have an optimal solution if the intermediate representation is a scheduled Control Flow Graph (CFG) in Static Single Assignment (SSA) Form. Register sharing allows program variables with non-overlapping lifetimes to reside in the same register. Without register sharing, each variable in the intermediate representation is stored in a separate register. An optimal solution to the register sharing problem minimizes the number of registers in the resulting datapath, yielding a more compact design with increased register utilization.

Register sharing has historically been modeled as a graph-coloring problem. For a program represented as a CFG in SSA form, the graph to be colored is proven to have an optimal solution with an \(O(|V| + |E|)\) time complexity. Results for 12 large procedures in SSA form show that optimal register sharing reduces the number of registers allocated to the design by 64.93% (42.83 registers) on average compared to linear-scan register allocation [1-3]. The per-benchmark percentage reduction ranged from 8.33% to 86.36% (1 to 87 registers).

II. HIGH-LEVEL SYNTHESIS AND REGISTER SHARING

Overview

High-level synthesis is the transformation of an intermediate representation of a computation into a structural description of a datapath that implements it. A high-level synthesis system contains many stages, including, but not limited to: resource allocation (computational, storage, and interconnect), scheduling, resource selection, resource sharing, binding operations to specific resource instances, and clock selection. These problems are interdependent, and different formulations exist for each depending on the order in which they are solved. Many of these problems are also NP-Complete.

A procedure body is represented as a CFG, a directed graph where nodes represent basic blocks—maximal sequences of straight-line code with no branches or branch targets interleaved—and edges represent transfers of control flow the end of one block to the beginning of another. Each basic block is represented by a directed acyclic graph (DAG) called a Data Flow Graph (DFG). A typical synthesis flow will begin by allocating computational resources (adders, multipliers, etc.) to the design and scheduling each operation on each resource. The scheduled DFG can then be represented by a list of sets of operations; the \(i^{th}\) set contains all of the DFG operations that were scheduled at the \(i^{th}\) time step.

The problem of register sharing is typically formulated and solved following resource allocation and scheduling. Let \(V\) be the set of variables in the program. An interference graph \(G = (V, E)\) is defined, where \((u, v) \in E\) indicates that variables \(u\) and \(v\) interfere—i.e., their lifetimes overlap, and thus require separate storage resources.

An independent set is a subset \(V' \subseteq V\) such that there are no edges between any pair of vertices in \(V'\). A \(k\)-coloring of \(G\) is a partition of \(V\) into \(k\) non-overlapping independent sets (color classes): \(\{C_1, C_2, ..., C_k\}\). A \(k\)-coloring can also be represented by a function \(f : V \rightarrow \{1, 2, ..., k\}\), defined such that \(f(v) = j\) is equivalent to the statement: \(v \in C_j\). A \(k\)-coloring ensures that for every edge \((u, v) \in E\), \(f(u) \neq f(v)\). The goal of register sharing is to color \(G\) with the fewest possible colors. Each color class represents a set of variables with non-overlapping lifetimes that share the same register.

In early synthesis systems, inputs were limited to DFGs. For a scheduled DFG, the interference graph for register sharing belongs to the class of interval graphs, which can be colored optimally in \(O(|V|\log|V|)\) time using the Left Edge Algorithm [2][3]; since all interval graphs are chordal, an \(O(|V| + |E|)\)-time coloring algorithm [4] could also be used.

Springer and Thomas [5] showed that chordal interference graphs arise during synthesis if certain conditions are imposed.
int foo(int x) {
    int y;
    if(x == 1) y = x+1;
    else y = x-1;
    return y;
}

Fig. 1. A small procedure (a) represented as a CFG (b) and shown in SSA form (c)

IV. STRICT PROGRAMS

A strict program [8] ensures that every variable is assigned a value before the variable is used in a computation along every possible path of program execution. Some languages impose strictness by definition. In Java, for example, all variables that are not explicitly initialized by the programmer are implicitly initialized to 0 by the compiler. Other languages, for example C/C++, do not impose strictness as a requirement. Most real-world programs are strict, regardless of language; however, non-strict programs do exist, and compilers and synthesis tools must be able to handle them.

A regular program [8] is a strict program in SSA form. A proof is given in the Appendix that an interference graph constructed for a regular program belongs to the class of chordal graphs. This is a notable result because graph coloring can be solved in $O(|V| + |E|)$ time for chordal graphs.

The theoretical result proven in this paper requires a regular program to ensure correctness. Non-strict programs explicitly violate SSA form, since some variables may exist that are not defined. Compilers such as GCC insert implicit definitions for all variables that are used but not defined. Implicit definitions can also be used to correct the case where a variable is defined on some (but not all) paths that converge at a φ-function. This imposes strictness on the intermediate representation, even if the input program is not strict. Throughout the duration of this paper, it is assumed that programs in SSA form are regular.

V. LIVENESS ANALYSIS FOR PROGRAMS IN SSA FORM

According to Cooper and Torczon [9, p. 630]: “variable $v$ is live at point $p$ if it has been defined along a path from the procedure’s entry to $p$ and there exists a path from $p$ to a use of $v$ along which $v$ is not redefined.” Liveness analysis is the process of computing the set of variables that are live at each
point in the CFG. To build an interference graph for a CFG in SSA form, a minor modification to standard liveness analysis is required. We have implemented and modified the liveness analyzer described by Cooper and Torczon [9, p. 437-447].

If $n$ is a basic block in a CFG, then $LIVEOUT(n)$ is the set of all such variables that are live upon exiting $n$. Intuitively, $LIVEOUT(n)$ contains those variables that are defined either in $n$ or some other node $n'$ from which $n$ is reachable, and are used in some CFG node $n''$ reachable from $n$. Liveness analysis computes $LIVEOUT(n)$ for every basic block in the program. To compute $LIVEOUT(n)$, two additional sets of variables are required: $UEVAR(n)$ and $VARKILL(n)$.

$UEVAR(n)$ is defined to be the set of upward-exposed variables in $n$. $UEVAR(n)$ contains all of the variables that are used in $n$ but are not defined in $n$—some block that precedes $n$ during program execution must define each variable in $UEVAR(n)$. $VARKILL(n)$ is the set of all variables that are defined in $n$. Both $UEVAR(n)$ and $VARKILL(n)$ can be constructed by a linear traversal of the operations in $n$.

Let $suc(n)$ be the set of successor CFG nodes of $n$. In other words, there must be some (conditional or unconditional) control transfer from $n$ to each node in $suc(n)$. $LIVEOUT(n)$ is defined recursively as follows:

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

During liveness analysis, Eq. (1) is repeatedly solved for each basic block in the program until stability is reached.

For programs in SSA form, let $preds(m)$ be the set of predecessors of CFG node $m$. Then for each CFG node $n \in preds(m)$, $UEVAR(m)$ is replaced with $UEVAR(m, n)$: the set of upwards-exposed variables from $m$ to $n$. This allows $m$ to expose different $\phi$-function parameters to predecessors on different incoming control paths—i.e. mutual exclusion. There may also be variables that are originally in the $UEVAR(m)$ set that are not defined by a $\phi$-function in $m$. These variables are still live upon entry to $m$. All such variables are added to every set $UEVAR(m, n)$ for each predecessor $n$ of $m$.

For a program in SSA form, the recursive equation for $LIVEOUT$ sets is rewritten as follows:

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

In Fig. 1 (c), $y_1 \in UEVAR(n_3, n_1)$ and $y_2 \in UEVAR(n_3, n_2)$. This yields the following live-out sets: $LIVEOUT(n_1) = \{y_1\}$ and $LIVEOUT(n_2) = \{y_2\}$.

**VI. CHORDAL GRAPHS**

This section summarizes relevant topics from the theory of chordal graphs. Historically, chordal graphs have also been called triangulated graphs and rigid circuit graphs. For a thorough treatment of the subject, the interested reader is referred to Golumbic’s textbook on algorithmic graph theory [10, Ch. 4]. Fig. 2 (a) and (b) show respective examples of chordal and non-chordal graphs.

There are, in fact, three equivalent criteria for chordality. Undirected graph $G$ is chordal if and only if

1. $G$ has no induced subgraph isomorphic to a $k$-hole
2. $G$ admits a perfect elimination order (PEO)
3. $G$ is the intersection graph of a set of subtrees of a tree

Let $G = (V, E)$ be an undirected graph. Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic to one another if there exists a one-to-one and onto bijection $g$: $V_1 \rightarrow V_2$, such that for all $u, v \in V_1$, $(u, v) \in E_1$ if and only if $(g(u), g(v)) \in E_2$.

A $k$-hole, defined by positive integer $k \geq 4$ is any graph isomorphic to $H = (V_0, E_0)$, where $V_0 = \{v_0, v_1, ..., v_k\}$, and $E_0 = \{(v_i, v_{i-1}), (v_{i-1}, v_i) \mid 0 \leq i \leq k-1\}$.

An elimination order $\sigma$ of graph $G = (V, E)$ is a numbering scheme that orders the vertices. $\sigma(v) = i$ indicates that $v$ is the $i$th element in the order. Once $\sigma$ has been computed, vertices are renamed so that $\sigma(v) = 1$.

Starting with an empty graph, $G$ can be built by adding vertices in the order of $\sigma$. When $v_i$ is added to $G$, all edges adjoining $v_i$ to vertices in $\{v_1, ..., v_i\}$ are added to $G$ as well. Let $G^{0} = (V^{0}, E^{0})$ represent the partially-built graph after adding vertices $\{v_1, ..., v_i\}$; in other words, $G^{0}$ is the subgraph of $G$ induced by $\{v_1, ..., v_i\}$.

Let $N(v)$ be the set of vertices adjacent to $v$ in $G$, and let $N[v] = N(v) \cup \{v\}$. For $v \in V^{0}$, let $N^{0}(v) = N(v) \cap V^{0}$ and $N^{0}[v] = N^{0}(v) \cup \{v\}$.

Vertex $v$ in $G$ is simplical if $N(v)$ is a clique—a complete subgraph. $\sigma$ is a PEO if $v_i$ is simplical in $G^{0}$ for $i = 1, 2, ..., n$. In Fig 2 (a), $<B, C, E, F, D, A>$ is a PEO. On the other hand, $<A, B, C, D, E, F>$ is not—$E$, the fifth vertex in the PEO, is not simplical since $N^{0}(E) = \{A, B, C\}$ is not a clique.

A PEO can be constructed for graph $G = (V, E)$ in $O(|V| + |E|)$ time using either Lexicographic Breadth First Search (Lex-BFS), or a somewhat simpler and faster implementation of Lex-BFS called Maximum Cardinality Search (MCS) [11]. Given a PEO, a chordal graph can be colored in $O(|V| + |E|)$ time [4]. Colors are assigned in PEO order: $<v_1, v_2, ..., v_n>$. 

![Fig. 2. A chordal (a) and non-chordal (b) graph; the 4-hole in (b) is shown with bold edges.](image-url)
The only vertices that constrain the color that is assigned to \( v_i \), are those in \( N^0(v_i) \). The first color not assigned to a vertex in \( N^0(v_i) \) is given to \( v_i \). If the colors are tested in some fixed order, a color will be found within \( |N^0(v_i)|+1 \) tries.

VII. Optimal Register Sharing for SSA Form Programs

First, we introduce the concept of dominance. A node \( n_i \) in a CFG dominates node \( n_j \) (denoted \( n_i \text{ dom } n_j \)) if every possible execution path from the entry node \( n_0 \) to \( n_j \) passes through \( n_i \). Trivially, \( n_0 \) dominates every other node in the CFG. The dominance relation satisfies the following three properties:

- Reflexivity: \( n_i \text{ dom } n_i \)
- Transitivity: \( n_i \text{ dom } n_j \land n_j \text{ dom } n_k \Rightarrow n_i \text{ dom } n_k \)
- Antisymmetry: \( n_i \text{ dom } n_j \land n_j \text{ dom } n_i \Leftrightarrow n_i = n_j \)

For instructions \( i_1 \text{ dom } i_2 \) if \( i_1 \)’s basic block dominates \( i_2 \)’s, or if \( i_2 \) precedes \( i_1 \) in the same basic block. For variables \( v_i \) and \( v_j \), \( v_i \text{ dom } v_j \) if the definition of \( v_i \) dominates the definition of \( v_j \).

Theorems 1 and 2 are due to Budimlić et al. [8]:

**Theorem 1.** If two variables \( u \) and \( v \) in a regular program interfere, then either \( u \text{ dom } v \) or \( v \text{ dom } u \).

**Theorem 2.** If two variables, \( v_j \) and \( v_k \), in a regular program interfere and the definition point of \( v_j \text{ dom } v_k \), then \( v_j \) is in the live-in set of the block in which \( v_k \) is defined, or both variables are defined in the same block.

The live-in set in Theorem 2, denoted \( \text{LIVEIN}(n) \), is the set of variables that are live at \( n \)’s entry point. Theorem 3 formally states that the interference graph for a regular program is chordal. A proof of Theorem 3 is given in the Appendix.

**Theorem 3.** Let \( G = (V, E) \) be an interference graph generated from a regular program. Then \( G \) is a chordal graph.

At IWLS in June 2005, Brisk et al. [5] stated a slightly weaker version of Theorem 3—that \( G \) is a perfect graph. The proof in that paper is sufficient for chordal graphs, a subclass of perfect graphs. Brisk corrected this oversight in his talk.

In a technical report, also in June 2005, Hack [13] published Theorem 3 independently from Brisk et al. In fact, an earlier proof had been discovered in 2002 by Alain Darte in a private conversation with Fabrice Rastello and email communication with Jean Ferrante. Darte’s proof used the subtree-intersection characterization of chordal graphs (Definition 3). Darte’s contribution was subsequently published in a technical report by Bouchez et al. [14] in August, 2005.

The technical reports by Hack and Bouchez et al. discuss Theorem 3 in the context of register allocation in software compilers (e.g. [1][19-20]), where the problem remains NP-Complete. Brisk et al. applied Theorem 3 to high-level synthesis, which has an analogous problem also called register allocation in past literature (e.g. [3][5-6]). The term “register sharing” is used here to refer to the problem in synthesis.

VIII. Computing an Optimal Coloring Without Building an Interference Graph

Given a CFG in SSA form, an optimal color assignment can be computed without explicitly constructing an interference graph. Let \( \text{LIVEDEF}(v) \) be the set of variables that are alive at the point in the program just before \( v \) is defined. By Theorem 1, for every variable \( u \in \text{LIVEDEF}(v), u \text{ dom } v \). A PEO can be constructed by creating an ordering \( \sigma \), where \( \sigma(v) > \sigma(u) \) for all \( u \in \text{LIVEDEF}(v) \). For some variable \( v_i \), such that \( \sigma(v_i) = i \), \( \text{LIVEDEF}(v_i) \) is identical to clique \( N^0(v_i) \) discussed in Section VI. Given \( \text{LIVEDEF}(v_i) \), the smallest color not assigned to a vertex in \( \text{LIVEDEF}(v_i) \) is then assigned to \( v_i \).

Pseudocode to perform color assignment without building an interference graph is given in Fig. 3. The dominator tree is traversed from the root toward the leaves, ensuring that each parent node in the tree is processed before all of its children. The instructions within each basic block \( n \) are traversed twice. First, \( n \) is traversed backward, in order to determine which

Algorithm: Chordal Color Assignment
Input: CFG in SSA Form

1. Let \( V \) be the set of SSA Variables
2. For \( i \leftarrow 1 \) to \( |V| \)
3. Free_Color\[i\] \leftarrow True
4. For each basic block \( n \), taken in dominance order
5. \( \text{LIVENOW} \leftarrow \text{LIVEOUT}(n) \)
6. For each instruction \( i^* \) in \( b \) taken in reverse order
7. \( \text{KILLS}(i^*) \leftarrow \emptyset \)
8. For each variable \( u \) used by \( i^* \)
9. If \( u \in \text{LIVENOW} \)
10. \( \text{LIVENOW} \leftarrow \text{LIVENOW} \cup \{u\} \)
11. \( \text{KILLS}(i^*) \leftarrow \text{KILLS}(i^*) \cup \{u\} \)
12. EndIf
13. EndFor
14. Let \( v \) be the variable defined by \( i^* \)
15. \( \text{LIVENOW} \leftarrow \text{LIVENOW} - \{v\} \)
16. EndFor
17. // At this point, \( \text{LIVENOW} = \text{LIVEIN}(n) \)
18. For each instruction \( i^* \) in \( b \) taken in order
19. Let \( v \) be the variable defined by \( i^* \)
20. \( j \leftarrow 1 \)
21. While( Free_Color\[j\] = False )
22. \( j \leftarrow j + 1 \)
23. EndWhile
24. Color\[v\] \leftarrow j
25. Free_Color\[j\] \leftarrow False
26. \( \text{LIVENOW} \leftarrow \text{LIVENOW} \cup \{v\} \)
27. For each variable \( u \in \text{KILLS}(i^*) \)
28. Free_Color[Color\[u\]] \leftarrow True
29. \( \text{LIVENOW} \leftarrow \text{LIVENOW} - \{u\} \)
30. EndFor
31. EndFor
32. EndFor

Fig. 3. Pseudocode for optimal color assignment without building an interference graph.
instructions represent the end of variable lifetimes. Let set $KILLS(i*)$ contain the variables whose lifetimes end at $i*$. After computing the $KILLS$ sets, a forward traversal of the list assigns colors to each variable. $LIVEDEF(v)$ is replaced in Fig. 3 with a set $LIVENOW$ that contains the variables that are live at each point in the CFG. Without $KILLS$ sets, the forward traversal could not determine if each variable remains in $LIVENOW$ or dies after each use.

Theorem 4 establishes the correctness of the algorithm in Fig 3; a proof can be found in the Appendix.

**Theorem 4.** Let $\alpha$ be the order in which variables in a regular program are assigned colors by Line 24 of Fig. 3, and let $G = (V, E)$ be the interference graph. Then $\sigma$ is a PEO of $G$.

**IX. EXPERIMENTAL RESULTS**

The Machine SUIF compiler [15] was used to test and evaluate the concepts and ideas presented in this paper. Programs are converted to SSA form and an interference graph is constructed. Since this study focuses on register sharing (high-level synthesis) rather than register allocation (compilers), spilling and copy coalescing are not considered. Copies are folded during SSA construction, and pruned SSA form [16] is used in all experiments. The experiments were performed on a 3.00 GHz Intel Pentium IV Processor with 512K cache and 1G RAM, running Mepis Linux.

Register sharing based on chordal graph coloring is compared to register sharing based on linear-scan allocation, which is summarized in Section X. The benchmarks used in this experiment are procedures written in C taken from SPEC INT 2000 [17] and Mediabench [18]. Each procedure in this experiment has an SSA-form interference graph with at least 5000 vertices. Results are not reported for smaller procedures. These procedures were selected because they include control flow constructs, and the runtime of the experiments are determined by the interference graph size.

Table I clearly illustrates that chordal graph coloring outperforms linear-scan. The difference in the number of registers allocated ranges from 1 (SHA1Transform) to 87 (expand_expr). On average, the difference in favor of chordal coloring was 42.83 registers per procedure. For smaller procedures, linear-scan is capable of producing (near-)optimal results. For procedures without conditional branches and loops, linear-scan is optimal.

The primary selling point of linear scan allocation is its runtime, not the quality of its results. Table II compares the runtime (in seconds) of linear-scan to two different implementations of chordal graph coloring: CHO-1 builds an interference graph using the technique described in Section V, computes a PEO using Maximum Cardinality Search, and then colors the interference graph optimally [4]; CHO-2 uses the procedure in Fig. 3 to compute an optimal color assignment without building an interference graph.

From Table II, the runtime of liveness analysis, which is required by all three coloring techniques, dominates the total runtime cost of coloring. The respective runtimes of LS and CHO-2, post-liveness analysis are reported as is. The post-liveness analysis runtime of CHO-1 is broken into two components: the runtime of interference graph construction, including memory allocation, and the runtime of computing a PEO and then coloring the chordal graph. Both LS and CHO-2 run much faster than CHO-1. Both components of CHO-1 run slower than LS and CHO-2 by at least one order of magnitude. CHO-2 runs faster than LS for all benchmarks.

The 12 interference graphs used in the experiments were much larger than the vast majority in both SPEC INT 2000 and Mediabench. The differences between CHO-2 and LS in terms of both solution quality and runtime are exacerbated.

**TABLE I**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>File</th>
<th>Procedure</th>
<th>Interference Graph</th>
<th>Number of Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>176.gcc</td>
<td>combine.c</td>
<td>try_combine</td>
<td>7359</td>
<td>112759</td>
</tr>
<tr>
<td>176.gcc</td>
<td>combine.c</td>
<td>simplifty_rtx</td>
<td>7018</td>
<td>41843</td>
</tr>
<tr>
<td>176.gcc</td>
<td>c-parse.c</td>
<td>ypparse</td>
<td>6260</td>
<td>61207</td>
</tr>
<tr>
<td>176.gcc</td>
<td>cse.c</td>
<td>fold_rtx</td>
<td>5879</td>
<td>67974</td>
</tr>
<tr>
<td>176.gcc</td>
<td>cse.c</td>
<td>cse_insn</td>
<td>6646</td>
<td>117368</td>
</tr>
<tr>
<td>176.gcc</td>
<td>expr.c</td>
<td>expand_expr</td>
<td>9753</td>
<td>63809</td>
</tr>
<tr>
<td>176.gcc</td>
<td>fold-const.c</td>
<td>fold</td>
<td>11135</td>
<td>83358</td>
</tr>
<tr>
<td>176.gcc</td>
<td>insn-receg.c</td>
<td>recog_5</td>
<td>5011</td>
<td>21197</td>
</tr>
<tr>
<td>176.gcc</td>
<td>jump.c</td>
<td>jump_optimize</td>
<td>6144</td>
<td>108078</td>
</tr>
<tr>
<td>186.crafty</td>
<td>option.c</td>
<td>Option</td>
<td>6193</td>
<td>7968</td>
</tr>
<tr>
<td>ghostscript</td>
<td>gdevstc.c</td>
<td>stc_put_params</td>
<td>5003</td>
<td>21464</td>
</tr>
<tr>
<td>pegwit</td>
<td>sha1.c</td>
<td>SHA1Transform</td>
<td>5663</td>
<td>46095</td>
</tr>
</tbody>
</table>
X. RELATED WORK

A. Register Allocation in Software Compilers

The first graph coloring register allocator for compilers was developed by Chaitin et al. [19][20]. This work included a proof that for every finite undirected graph $G$, a CFG could be constructed such that liveness analysis would build an interference graph isomorphic to $G$. The CFG, however, was not strict, and the proof predated the discovery of SSA form.

For procedures without loops or branches, both LS and CHO-1/CHO-2 will produce optimal colorings.

Y. CONCLUSION

The problem of register sharing on a CFG in SSA form has been shown to have a polynomial-time solution based on chordal graph coloring. By sidestepping the construction of the interference graph, optimal register sharing can run faster than linear-scan allocation. Register sharing will reduce the overall area of the design and increase the utilization of registers. If resource $R$ is connected to $k$ registers $r_1, r_2, \ldots, r_k$, and $y$ can only be merged together if they do not interfere.
APPENDIX

Proof of Theorem 3.

Let $V' \subseteq V$, $|V'| = k \geq 4$. We show that $G'$, the subgraph of $G$ induced by $V'$, is not isomorphic to a $k$-hole. Assume to the contrary that $G'$ is isomorphic to a $k$-hole. Let $s$ be the unique source and $t$ be the unique sink in $G'$, as defined, then $s, t \in V'$.

First, we argue that $G'$ must be acyclic. Since $G'$ is a hole, the only cycle would be the sequence $s, v_1, \ldots, v_{k-1}, t$, its reverse, or a circular shift thereof. Without loss of generality, let $v_k dom v_1$; however, path $s, v_1, \ldots, v_{k-1}, t$ coupled with the transitivity of the domination relation indicates that $v_i dom v_0$, violating antisymmetry. Therefore $G'$ must be acyclic.

In a directed graph, $N(v)$ is divided into incoming and outgoing adjacent vertices. $N_{in}[v] = \{u | (u, v) \in E\}$ and $N_{out}[v] = \{w | (v, w) \in E\}$. Since the original undirected graph $G$ is a hole, every vertex $v$ satisfies the following property: $|N_{in}[v]| + |N_{out}[v]| = 2$. A source is defined to be a vertex $s$ such that $|N_{in}[s]| = 0$. A sink is defined to be a vertex $t$ such that $|N_{out}[t]| = 0$. It is a fundamental result from graph theory that every DAG has at least one source and at least one sink.

Let $s = v_i$ and $t = v_j$ be a respective source and sink in $G'$. Without loss of generality, suppose that $i < j$. If $j > i$, simply reverse all of the edges in $G'$ and swap $s$ with $t$. Antisymmetry ensures that the same vertex cannot be both a source and a sink in $G'$.

We do not know whether $s$ and $t$ are the unique source and sink, or if there are others. If $s$ and $t$ are unique, then there exist two directed paths from $s$ to $t$: $P_1 = \langle s, v_{i+1}, v_{i+2}, \ldots, v_j \rangle$ and $P_2 = \langle s, v_{i+1}, v_{i+2}, \ldots, v_k, v_{k-1}, v_{k-2}, \ldots, t \rangle$, illustrated in Fig. 4.

We show by induction that $P_t$ is a directed path in $G$. For the basis, $\langle s, v_{i+1} \rangle$ is trivially a directed path since $s$ is a source. Now, let $\langle s, v_{i+1}, \ldots, v_{i+\alpha} \rangle$, where $1 \leq \alpha \leq j - i - 1$ be a directed path from $s$ to $v_{i+\alpha}$. Now, consider $v_{i+\alpha+1}$.

Assume to the contrary that $(v_{i+\alpha+1}, v_{i+\alpha})$ defines the direction of the edge between $v_{i+\alpha+1}$ and $v_{i+\alpha}$; consequently, $v_{i+\alpha} dom v_{i+\alpha+1}$. By Theorem 2, either $v_{i+\alpha} \in \text{LIVEIN}(n)$ or $v_{i+\alpha}$ is defined in $n$.

By the induction hypothesis, the direction of edge $(v_{i+\alpha+1}, v_{i+\alpha})$ has already been resolved; consequently, $v_{i+\alpha} dom v_{i+\alpha+1}$. Once again, Theorem 2 shows that either $v_{i+\alpha} \in \text{LIVEIN}(n)$ or $v_{i+\alpha+1}$ is defined in $n$. In either case, $v_{i+\alpha}$ and $v_{i+\alpha+1}$ must both be live at the definition point of $v_{i+\alpha}$ in $n$, and $v_{i+\alpha}$ and $v_{i+\alpha+1}$ interfere. Since a hole must contain at least 4 vertices, and edges $(v_{i+\alpha}, v_{i+\alpha+1})$, $(v_{i+\alpha}, v_{i+\alpha+2})$, and $(v_{i+\alpha}, v_{i+\alpha+3})$ are known to exist, the existence of edge $(v_{i+\alpha+1}, v_{i+\alpha+2})$ would violate the assumption that $G'$ is a hole—a contradiction. Consequently, $(v_{i+\alpha}, v_{i+\alpha+1})$ must be the direction of the edge, ensuring that $\langle s, v_{i+1}, \ldots, v_{i+\alpha} \rangle$ is a directed path from $s$ to $v_{i+\alpha+1}$.

This establishes $P_t$ is a directed path from $s$ to $t$. An identical argument proves that $P_2$ is also a directed path from $s$ to $t$. Together, the existence of directed paths $P_1$ and $P_2$ ensure that there is exactly 1 source and 1 sink in $G'$. Now, consider sink $t$. Since $t$ is a sink, $\text{NIN}[t] = \{v_{i+1}, v_{i+2}\}$.

Consequently, $v_{i+1} dom t$ and $v_{i+2} dom t$. If $n$ is the basic block in which $t$ is defined, then $v_{i+2} \in \text{LIVEIN}(n)$ or $v_{i+1}$ is defined in $n$; likewise, both $v_{i+1} \in \text{LIVEIN}(n)$ or $v_{i+2}$ is defined in $n$. Both $v_{i+1}$ and $v_{i+2}$ interfere at the definition point of $v_{i+2}$. Once again, a hole known to contain edges $(v_{i+1}, v_{i+2})$ and $(v_{i+2}, v_{i+3})$ cannot also contain an edge $(v_{i+1}, v_{i+2})$. This contradicts the assumption that $G'$ is isomorphic to a $k$-hole.

\[\square\]

Proof of Theorem 4.

Consider basic block $n$. Let $\text{DEFS}[n]$ be the set of variables defined in $n$. For $u \in \text{LIVEIN}(n)$, $v \in \text{DEFS}[n]$, $\sigma(u) < \sigma(v)$ since basic blocks are processed in dominance order. For $v, w \in \text{DEFS}[n]$, $\sigma(v) < \sigma(w)$ if and only if $v$ is defined before $w$ in $n$ since Line 24 occurs within a forward traversal of $n$. Then for each variable $x \in \text{LIVEDEF}[v]$, $\sigma(x) > \sigma(x)$.

Now, consider interference edge $(u, v) \in E$. By Theorem 2, either (1) $u dom v$ and $u \in \text{LIVEDEF}[v]$, or (2) $v$ dom $u$ and $\sigma(v) < \sigma(u)$. Let $G' = (V', E')$ be the subgraph of $G$ induced by $V' = \{u, v\} \in E \mid \sigma(u) < \sigma(v)\}$. Then $u, v \in E'$ if and only if $u \in \text{LIVEDEF}[v]$. Since $\text{LIVEDEF}[v]$ is a clique, $v$ is simplicial in $G'$ and $\sigma$ is a PEO of $G$.

\[\square\]

ACKNOWLEDGMENT

The following individuals provided insightful comments on this work: Daniel Berlin, Alain Darte, Sebastian Hack, Jens Palsberg, Fernando M. Q. Pereira, Fabrice Rastello, Leon Stok, and the anonymous reviewers.

REFERENCES


[17] Majid Sarrafzadeh (M’87–SM’92–F’96) received his B.S., M.S. and Ph.D. in 1982, 1984, and 1987 respectively from the University of Illinois at Urbana-Champaign in electrical and computer engineering.


[19] Dr. Sarrafzadeh is on the editorial board of the VLSI Design Journal, an Associate Editor of ACM Transactions on Design Automation (TODAES) and an Associate Editor of IEEE Transactions on Computer-Aided Design (TCAD). Dr. Sarrafzadeh has collaborated with many industries in the past fifteen years including IBM and Motorola and many CAD industries and was the architect of the physical design subsystem of Monterey Design Systems’ main product. He is a co-founder of Hier Design, Inc.